Rational Environment
Reference Manual

Data and Device Input/Output (DIO)

Document Control Number: 8001A-27

Rev. 1.0, October 1985
Rev. 2.0, December 1985
Rev. 3.0, May 1986
Rev. 4.0, July 1987 (Delta)

Note the Reader's Comments form on the last page of this book, which requests the user's evaluation to assist Rational in preparing future documentation.

Rational
1501 Salado Drive
Mountain View, California 94043

Contents

RATIONAL

How to Use This Book

The Data and Device Input/Output (DIO) book of the *Rational Environment Reference Manual* contains reference information describing some of the I/O packages provided by the Rational Environment™ for manipulating binary files, devices, and editor windows. This includes reference information on the Ada®-predefined packages Direct_Io, Sequential_Io, and Io_Exceptions, as well as information on Rational®-developed I/O packages. Note that packages for performing I/O on text files are documented in the Text Input/Output (TIO) of the *Rational Environment Reference Manual.* The reference entries for package !Io.Io_Exceptions are duplicated in both DIO and TIO, because these exceptions can be raised by any of the I/O packages.

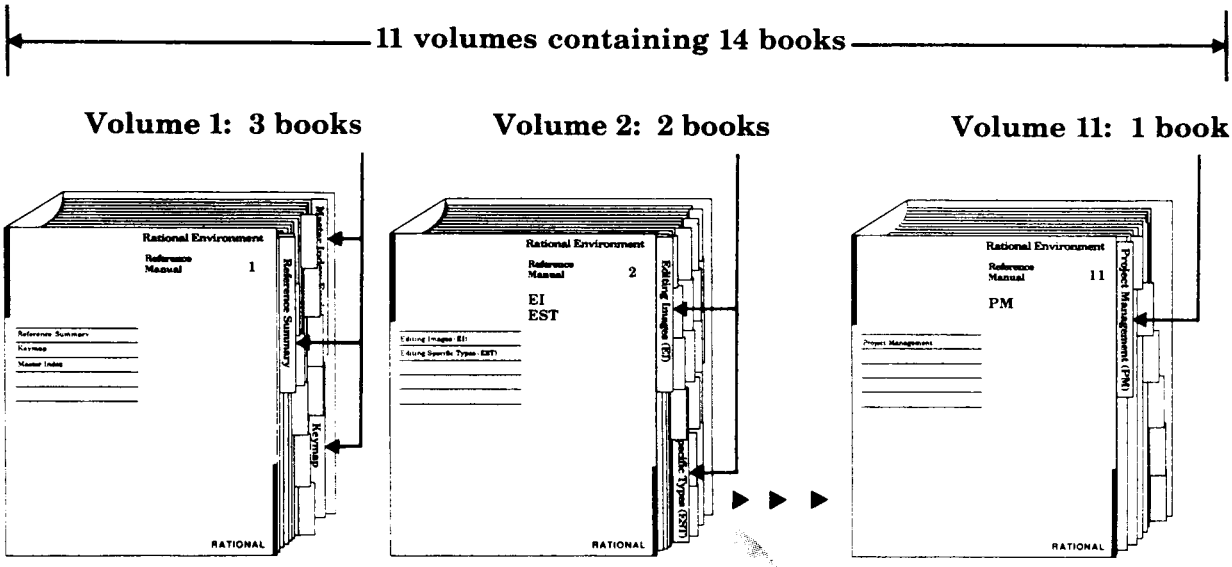
Organization of the Reference Manual

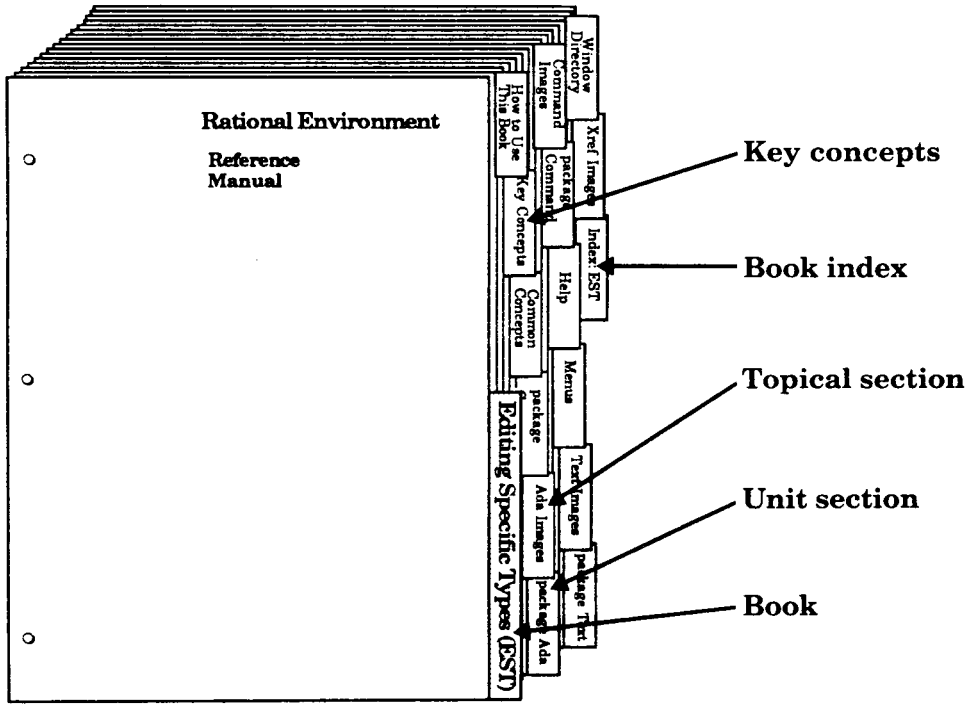
The *Rational Environment Reference Manual* (Reference Manual for brevity) includes the following volumes (see accompanying illustration):

1	Reference Summary Keymap Master Index
2	Editing Images (EI) Editing Specific Types (EST)
3	Debugging (DEB)
4	Session and Job Management (SJM)
5	Library Management (LM)
6	Text Input/Output (TIO)
7	Data and Device Input/Output (DIO)
8	String Tools (ST)
9	Programming Tools (PT)
10	System Management Utilities (SMU)
11	Project Management (PM)

Each *volume* of the Reference Manual contains one or more *books* separated by large colored tabs. Each book contains information on particular features or areas of application in the Environment. The abbreviation for the name of each book (for example, EI for Editing Images) appears on the binder cover and spine, and this abbreviation is used in page numbers and cross-references. The books grouped into one volume are not necessarily logically related.

Organization of the *Rational Environment Reference Manual*





A sample book

The Reference Manual provides reference information organized to efficiently answer specific questions about the Rational Environment. The *Rational Environment User's Guide* complements this manual, providing a user-oriented introduction to the facilities of the Environment. Products other than the Rational Environment (for example, Rational Networking—TCP/IP or Rational Target Build Utility) are documented in individual manuals, which are not part of the Reference Manual.

Volume 1

Volume 1, intended to be used as a quick reference to the resources provided by the Environment, contains the following books:

- **Reference Summary**: The Reference Summary contains the full Ada specification for each unit in the standard Environment. The unit specifications are organized by their pathnames. The World ! section provides a list of the units in the library system of the Environment and the manual/book in which they are documented.
- **Keymap**: The Rational Environment Keymap presents the standard Environment key bindings, organized by topic and by command name. The topical section includes both a quick reference for commonly used commands and a more detailed reference for key bindings.
- **Master Index**: The Master Index combines all of the index information for each of the books in the Reference Manual.

Volumes 2–11

Each book in Volumes 2–11 begins with a colored tab on which the name of the book appears. Each book typically contains the following sections:

- **Contents**: The table of contents provides a complete list of all the units in the book and their reference entries.
- **Key Concepts section**: Most of the books contain a section describing key concepts that pertain to all of the Environment facilities documented in that book. This section is located behind its own tab after the table of contents.
- **Unit sections**: Each of the commands, tools, and so on has a declaration within an Ada compilation unit (typically a package) in the Environment library system. For each unit, there is a section that contains reference entries for the declarations (for example, procedures, functions, and types) within that unit. Each section is preceded by a tab.

  The sections for units are alphabetized by the simple names of the units. For example, the section for package !Tools.String_Utilities is alphabetized under String_Utilities.

  For many units, introductory material and/or examples specific to the unit appear after the section tabs.

  Within the section for a given unit, the reference entries describing the unit's declarations are organized alphabetically after the section introduction. Appearing at the top of each page in a reference entry are the simple name of the given declaration and the fully qualified pathname of the enclosing unit.

- **Explanatory/topical sections**: Like the unit sections, explanatory/topical sections are preceded by tabs, and they are alphabetized with the unit sections. The topical sections, such as Help, located in Editing Specific Types (EST), discuss Environment facilities.
- **Index**: Preceded by a tab, the Index appears as the last section of each book. It contains entries for each unit or declaration, along with additional topical references. Each book index covers only the material documented in that particular book. The Master Index (in Volume 1) provides entries for the information documented in all the books within the Reference Manual.

  Italic page numbers indicate the page on which the primary reference entry for a declaration appears; nonitalic page numbers indicate key concepts, defined terms, cross-references, and exceptions raised.

Suggestions for Finding Information

The following suggestions may help you in finding various kinds of information in the documentation for Rational's products.

Learning about Environment Facilities

If you are a novice user starting to use the Environment, consult the *Rational Environment User's Guide.*

If you are familiar with the Environment but are interested in learning about the Environment's library-management commands, for example, you might start by scanning the specifications for these units in the Reference Summary to get an idea of the kinds of things these tools can do. You should also look at the Key Concepts for the particular book, which describes important concepts and gives examples.

It may also be useful to glance through the introductions provided for some of the units in the book. These introductions, located immediately after the tabs for the units, often contain helpful examples.

Finding Information on a Specific Item

If you know the name of the item and the book in which it is documented, consult either the table of contents or the index for that book. You can also turn through the pages of the book using the names and pathnames of the reference entries to locate the entry you want. Remember that the reference entries for a unit are organized alphabetically within the unit, and the units are organized alphabetically by simple name within the book.

If you know the simple name of the entry but do not know the book in which it is documented, look in the Master Index (in Volume 1) to find the book abbreviation and page number.

If you know the pathname of the entry but do not know the book in which it is documented, the World ! section of the Reference Summary (in Volume 1) provides a map of the units in the library system of the Environment and the books in which they are documented.

If you cannot find an item in the Master Index, the item either is not documented or is documented in the manuals for a product other than the Rational Environment (for example, Rational Networking—TCP/IP or Rational Target Build Utility). If you know the pathname, consult the World ! section of the Reference Summary to determine whether that item is documented and in which manual.

Using the Index

The index of each book contains entries for each unit and its declarations, organized alphabetically by simple name. When using the index to find a specific item, consult the italic page number for the primary reference for that item. Nonitalic page numbers indicate key concepts, defined terms, cross-references, and exceptions raised.

Viewing Specifications On-Line

If you know the pathname of a declaration and want to see its specification in a window of the Rational Environment, provide its pathname to the Common.Definition procedure—for example, `Definition ("!Commands.Library");`. If you know the simple name of the unit in which the declaration appears, in most cases you can use searchlist naming as a quick way of viewing the unit—for example, `Definition ("\Library");`.

Using On-Line Help

Most of the information contained in the reference entries for each unit is available through the on-line help facilities of the Environment. Press the Help on Help key or consult the *Rational Environment User's Guide* or the *Rational Environment Reference Manual*, EST, Help, for more information on using this on-line help facility.

Cross-Reference Conventions

The following conventions are used in cross-references to information:

- **Specific page/book**: For references to a specific place in a specific book, the book abbreviation is followed by the page number in the book (for example, LM-322). If the book abbreviation is omitted, the current book is implied (for example, the page numbers in the table of contents for a book do not include the book prefix).
- **Declaration in same unit**: References to the documentation for a declaration in the same unit are indicated by the simple name of the desired declaration. For example, within the reference entry for the Library.Copy procedure, a reference to the Library.Move procedure would be simply "procedure Move." Note that if there are nested packages in the unit, references to nested declarations use qualified pathnames.
- **Declaration in different unit, same book**: References to the documentation for a declaration in another unit are indicated by the qualified pathname of the desired declaration. For example, within the reference entry for the Library.Copy procedure, a reference to the Compilation.Delete procedure would be "procedure Compilation.Delete."

- **Declaration in different book**: References to the documentation for a declaration in another book are indicated by the addition of the abbreviation for that book. For example, within the reference entry for the Library.Copy procedure, a reference to the Editor.Region.Copy procedure in the Editing Images book would be "EI, procedure Editor.Region.Copy."

References to specific declarations in the library system of the Rational Environment (not the documentation for them) are typically indicated by fully qualified pathnames—for example, "procedure !Commands.Library.Copy." When the context is clear, however, a shorter name will be used. If the unit in which the declaration appears is undocumented, you may want to see its explanatory comments to understand what it does. To see these comments, either look at the unit's specification in the Reference Summary or view it on-line using the Rational Environment.

Feedback to Rational: Reader's Comments Form

Rational wants to make its documentation as useful and error-free as possible. Please provide us with feedback. The last page of each book contains a Reader's Comments form that you can use to send us comments or to report errors. You can also submit problem reports and make suggestions electronically by using the SIMS problem-reporting system. If you use SIMS to submit documentation comments, please indicate the manual name, book name, and page number.

Key Concepts

Data and Device Input/Output (DIO) contains reference information describing some of the I/O packages provided by the Rational Environment for manipulating binary files, devices, and editor windows. This includes reference information on the Ada-predefined packages Direct_Io, Sequential_Io, and Io_Exceptions, as well as information on the Rational-developed I/O packages Polymorphic_Sequential_Io and Window_Io.

Text Input/Output (TIO) contains reference information describing the I/O packages for manipulating text files. Text files are files that contain ASCII characters, which are of the Character type intended for viewing, editing, and so on.

Package Window_Io is used to perform I/O to editor windows.

Package Polymorphic_Sequential_Io provides facilities similar to those of package Sequential_Io, except that the I/O stream can be polymorphic—that is, the same stream can pass values of one or more different types. This capability is most useful when writing applications that must write and read values of multiple types to and from the same file or device—for example, database applications or various kinds of development tools. This capability is achieved by exporting a nested generic unit that must be instantiated for each type of data in the stream. As with all the I/O facilities, attempting to read data into an object of a type different from the data that are available for input may result in an exception being raised. Thus, it is the responsibility of the application to keep track of the order in which data of a specific type are to be written and read.

Files

The I/O packages manipulate information in objects stored in the library system of the Rational Environment. This includes files, Ada units, and devices such as windows and terminals. Since the Rational Environment offers a richer definition of the file than does the *Reference Manual for the Ada Programming Language*, in the description of all the I/O facilities in the Rational Environment, the term *files* may be used to denote any one of these entities.

Files that are objects of the file class in the library system of the Environment can be read from or written to. In the Rational Environment, a file is identified in a library display with an entry of the form:

```
name : file;
```

where name is the identifier of the simple name of the file. Files can exist only in libraries—that is, directories or worlds. Files can be created, opened, closed, deleted, and otherwise read from/written to by any of the Environment I/O packages. EST, package Text, provides facilities for text-specific editing of files, and a file append operation is available in LM, package File_Utilities, and in TIO, package Io. It is common for a file to be created by the user with the facilities of package Text (EST) and then later read by the I/O packages discussed in this section.

Files thus provide the conventional notion of file storage. When a file is modified using the Rational Editor, changes to the file are not preserved until the file is committed. When a file is created or modified from a program, the updated value of the object is committed only when the file is closed. Thus, if a program does not explicitly close a file, the permanent contents of the file are unchanged by the execution of the program. This may be the intended result, but caution is warranted, especially in error situations in which exception handlers must determine whether to save the contents of a file by closing it. Not closing the file effectively abandons the changes made by the program.

Ada units are generally created through normal program development using the resources of the Rational Editor. Since Ada units can be read as streams of characters, the I/O facilities discussed in this book can be used to read this image. Ada units cannot be written directly. However, facilities exist in the Rational Environment for transforming a file into an Ada unit (see, for example, LM, procedure Compilation.Parse).

Files and Ada units are subject to the standard read/write synchronization protocols used throughout the Rational Environment. This synchronization permits multiple jobs to have simultaneous access to the same file for reading, but it allows only a single job to write to a file at any instant in time (with no readers allowed while a writer has the file open). Attempting to gain access to an object in a manner that violates this protocol results in the Io_Exceptions.Use_Error exception being raised. Attempting to open an Ada unit for writing also results in the Use_Error exception being raised.

Devices and Windows

The Rational Environment supports I/O to or from several *devices*, including windows and terminals. In general, all of the I/O packages documented in this book can use any of these devices. I/O to tapes is permitted and is provided by package Tape (SMU). The exact effect of I/O with a particular device is, of course, unique to that device, and is explained in the following paragraphs.

For each user session, one or more windows can be created to provide a medium for the files Standard_Input and Standard_Output, as defined in TIO, packages Text_Io and Io. Multiple windows are created when more than one job is simultaneously performing output. These windows are given names corresponding to the name of the job that is currently accessing them, or that accessed them most recently, and is of Text type. This window can be moved or expanded, and its contents can

be cut, copied, and otherwise manipulated with the Rational Editor commands, as explained in Editing Images (EI) and Editing Specific Types (EST), package Text, in the *Rational Environment Reference Manual.* In general, this window automatically pops up when I/O is requested of the standard files. Output to Standard_Output appears in the window as characters (with control characters highlighted in a special font). Input requested from Standard_Input is denoted with the typical editor prompt, with the name */input/*. The usual editing paradigm offered by the Rational Editor applies, so input can be typed ahead, edited, and even copied from other windows. Input is not sent to the waiting program until it is committed.

Session I/O windows optionally accumulate all I/O to standard files during one session, so the windows can be used to keep scripts of program interaction. Between jobs, I/O to these windows is separated by a job separator. The files Standard_Input and Standard_Output are automatically created at the start of each job and are automatically closed at the end of each job. If more than one job is initiated in a single session, and each job uses the resources of Standard_Input or Standard_Output, additional windows are created as necessary. If a job is executed and a session does not exist, Standard_Input and Standard_Output map to files with the names Standard_Input and Standard_Output, respectively. These files are created in the default context of the job that initiated the I/O.

Package Io introduces the notion of a Standard_Error file. This file maps to the Rational Editor Message window, so it typically is used to provide a common error-reporting mechanism among tools. If a job is executed and a session does not exist, Standard_Error maps to a file with the name Standard_Error, created in the default context of the job that initiated the I/O.

A programmatic interface for performing I/O to windows is provided with package Window_Io. In this case, the file abstraction is associated with an image. This image is displayed in a window on the terminal screen. Interfaces are provided to open images, put characters to any part of the image, get characters from the image, and close or delete the image when finished.

I/O also can be initiated directly to terminals using any of the I/O packages. Files can be opened using a name (of String type) in the form !Machine.Devices.Terminal_n, where *n* is an integer corresponding to a physical port on the processor. The exact names available on any particular machine can be found in !Machine.Devices. Once the file is opened, I/O can proceed as with any other file. Of course, the effect of the I/O depends on the nature of the physical device attached to the port. Physical devices other than the Rational Terminal can be attached to any port. If a program attempts to open a terminal that is already assigned to a job, the Io_Exceptions.Use_Error exception is raised.

Note that other lower-level operations for performing I/O on terminals that are logged in are available from package !Io.Device_Independent_Io, using the operations in package !Io.Terminal_Specific. They are not documented in the *Rational Environment Reference Manual.*

Safe Types

Packages Sequential_Io, Direct_Io, and Polymorphic_Sequential_Io can be used to perform I/O on any *safe type.* A safe type is any type that does not contain access types or task types in any of its components. If a create or an open operation is attempted with an instantiation of an unsafe type, the Io_Exceptions.Use_Error exception will be raised. If a read or a write operation is attempted with an instantiation of an unsafe type, the Io_Exceptions.Data_Error exception will be raised.

File Handles

File handles are used for performing operations on files within Ada programs using the facilities provided by the I/O packages. When a file is opened or created, a file handle is returned. This handle is then used to refer to the file when calling the subprograms in the I/O packages.

The following is an example of a program that reads the lines from a text file named !Users.Blb.A_Text_File and displays them in the output window. The program first opens the file which returns a file handle. This file handle is then used for reading the lines from the file and checking for an end of file condition.

```
with Io;
procedure Display_File is

    -- This program reads lines from a text file and displays them
    -- in the output window.

    File_Handle : Io.File_Type;
    -- This is the object that will contain the file handle.

begin

    Io.Open (File => File_Handle,
             Mode => Io.In_File,
             Name => "!users.blb.a_text_file");
    -- Opens the named file for reading and returns a file handle for
    -- performing I/O operations on that file within this program.

    while not Io.End_Of_File (File_Handle) loop
        declare
            Line : constant String := Io.Get_Line (File_Handle);
            -- Reads a line from the file.
        begin
            Io.Put_Line (Line);
            -- Writes the line to the output window (Standard_Output).
        end;
    end loop;

end Display_File;
```

Filenames

Filenames supplied to the Create and Open procedures in the various I/O packages can be any legal Environment object name that uniquely identifies an object. Such names, for example, can contain wildcards and so on as long as the name can be resolved to a single object. Note that special names (for example, "<SELECTION>") can also be used to designate the name of a file. For more information on naming objects, see SJM, Key Concepts.

Access Control

The Rational Environment provides access-control mechanisms that can be used to restrict the access that users and programs have to the objects in the library system. The operations provided in the I/O packages are subject to these access controls.

The access specified in Table 1-1 is required for performing I/O operations on files. If the required access does not exist, the Io_Exceptions.Use_Error exception will be raised by the attempted operation. See LM, Key Concepts, for more information on access control.

Table 1-1. Access Required for I/O Operations.

Operation	Access Required
All operations	Read access for all worlds enclosing the file
Creating a file	Create access to the world in which the file is to be created
Deleting a file	Read access to the file
Opening a file for reading (mode In)	Read access to the file
Opening a file for writing (mode Out)	Write access to the file

Concurrency

The execution of any command or subprogram in the Rational Environment constitutes a *job*. Within a job, there may be several tasks that use I/O resources. If multiple tasks all share that same file handle, I/O may be arbitrarily interleaved and the results can be unpredictable. Thus, the I/O resources documented in this book may not offer or imply synchronization of the I/O activity. The Rational Environment does provide synchronization of I/O among different jobs, as discussed in "Devices and Windows," above.

Representations of Terminators

Since packages Text_Io and Io (TIO) observe the abstraction required by the *Reference Manual for the Ada Programming Language* of files containing line, page, and file terminators, it is sometimes useful to permit the user to simulate these terminators when creating or reading text files using the facilities in DIO. In the Environment, the line terminator is denoted by the character Ascii.Lf, the page terminator is denoted by the character Ascii.Ff, and the end-of-file terminator is implicit at the end of the file. A line terminator directly followed by a page terminator is compressed to the single character Ascii.Ff. Also, the line and page terminators preceding the file terminator are implicit and do not appear as characters in the file. For the sake of portability, programs should not depend on this representation, although it can be necessary to use this representation when importing text files from another system or exporting text files from the Rational Environment.

Exceptions

Note that although most of the I/O packages contain renaming declarations for the exceptions defined in package Io_Exceptions, descriptions of these renaming declarations are omitted from the packages. Refer to the descriptions of the exceptions in the reference entries for package Io_Exceptions.

The Rational Environment provides additional information about exceptions raised by the I/O packages. This information, which describes why a given exception occurred, is typically displayed in parentheses after the exception name. See the reference entries for the exceptions in package Io_Exceptions for descriptions of this additional information.

Error Reactions

When errors are discovered in a command, the command can respond by:

- Ignoring the error and trying to continue
- Issuing a warning message and trying to continue
- Raising an exception and abandoning the operation

For each job, the Environment maintains a default action for commands in package Profile (SJM) to take if an error occurs. There are commands for specifying and displaying the default error reaction for a job. Regardless of the default error reaction, any error reaction can be specified for any command.

The Environment has three default specifications for the profile it should use when responding to errors in a command. These are "<PROFILE>", "<SESSION>", and "<DEFAULT>", which refer, respectively, to the job response profile, the session response profile, and the default profile returned by the Profile.Default_Profile function.

generic package Direct_Io

This package provides the capabilities for Direct_Io as required by the *Reference Manual for the Ada Programming Language*, Chapter 14. It provides facilities for direct I/O upon files whose components are of the same (nonlimited) type. This type must be a safe type—that is, any type that does not contain access types or task types in any of its components. If a create or an open operation is attempted with an instantiation on an unsafe type, the Io_Exceptions.Use_Error exception will be raised.

The fundamental abstraction provided by package Direct_Io is the File_Type type. Objects of this type are file handles that can be mapped to files. A file is viewed as a set of elements occupying consecutive positions in linear order; a value can be transferred to or from an element of the file at any selected position. The position of an element is specified by its index. The first element, if any, has index 1; the index of the last element, if any, is called the current size (which is 0 if there are no elements).

procedure Close

```
procedure Close (File : in out File_Type);
```

Description

Severs the association between the file handle and its associated file.

Parameters

`File : in out File_Type;`
Specifies the handle for the file.

Errors

If the file is not open, the Io_Exceptions.Status_Error exception is raised.

type Count

```
type Count is new Integer range 0 .. Integer'Last / Element_Type'Size;
```

Description

Defines the valid range of direct file index positions.

procedure Create

```
procedure Create (File : in out File_Type;
                 Mode :         File_Mode := Inout_File;
                 Name :         String    := "";
                 Form :         String    := "");
```

Description

Establishes a new file with the given name and associates this file with the specified file handle.

The specified file is left open.

Parameters

`File : in out File_Type;`

Specifies the handle for the file.

`Mode : File_Mode := Inout_File;`

Specifies the access mode for which the file is to be used.

`Name : String := "";`

Specifies the name of the file to be created. A null string for the Name parameter specifies a file that is not accessible after the completion of the main program (a temporary file).

`Form : String := "";`

Currently, the Form parameter, if specified, has no effect.

Restrictions

Files can be created only in directories or worlds.

Errors

If the specified file handle is already open, the Io_Exceptions.Status_Error exception is raised.

The Io_Exceptions.Name_Error exception is raised under any of the following conditions:

- The filename does not conform to the syntax of a name.
- An object of a nonfile class with the same name as the filename already exists in the context in which the creation is attempted.
- The context in which the creation is attempted cannot contain files. Files are allowed only in directories or worlds.

The Io_Exceptions.Use_Error exception is raised under any of the following conditions:

- The Element_Type type is unsafe (that is, it contains access or task types).
- The file cannot be opened with the specified mode.
- Another job has locked the file.
- The executing job does not have create access.

procedure Delete

```
procedure Delete (File : in out File_Type);
```

Description

Deletes the file associated with the specified file handle.

The specified file is closed, and the file ceases to exist.

Parameters

```
File :  in out File_Type;
```
Specifies the handle for the file.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

The Io_Exceptions.Use_Error exception is raised under any of the following conditions:

- The Environment does not support deletion on the file.
- The executing job does not have the access rights required to delete the file.
- Another job has locked the file.

generic formal type Element_Type

```
type Element_Type is private;
```

Description

Defines the type of the items that form the files for each particular instantiation.

Restrictions

The type used to instantiate the generic must be a nonlimited type as well as a safe type. Specifically, the type cannot be an access or task type and cannot contain components that are access or task types.

If unsafe types are used for the instantiation, subsequent create or open operations will raise the Io_Exceptions.Use_Error exception.

function End_Of_File

```
function End_Of_File (File : File_Type) return Boolean;
```

Description

Returns true if the current index exceeds the size of the file; otherwise, the function returns false.

This function operates on a file of the In_File or Inout_File mode.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`return Boolean;`
Returns true if the current index exceeds the size of the file; otherwise the function returns false.

Errors

If the file is opened with the Out_File mode, the Io_Exceptions.Mode_Error exception is raised.

type File_Mode

```
type File_Mode is  (In_File, Inout_File, Out_File);
```

Description

Specifies the mode of access for which a file is open.

In_File denotes a file with read-only access, Out_File denotes a file with write-only access, and Inout_File denotes a file with read/write access.

type File_Type

```
type File_Type is limited private;
```

Description

Defines the type of the file handle unique to each instantiation of the package.

Objects of this type are file handles that can be mapped to external files.

function Form

```
function Form (File : File_Type) return String;
```

Description

Returns the null string ("") in all cases.

When the Form parameter to the Create and Open procedures is supported in the future, the Form value provided to the call to the Open or Create procedure will be returned.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`return String;`
Returns the null string ("") in all cases.

Errors

If the file is not open, the Io_Exceptions.Status_Error exception is raised.

References

procedure Create

procedure Open

function Index

```
function Index (File : File_Type) return Positive_Count;
```

Description

Returns the current index of the specified file.

This function operates on a file of any mode.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`return Positive_Count;`
Returns the current index of the specified file.

function Is_Open

```
function Is_Open (File : File_Type) return Boolean;
```

Description

Returns true if the file handle is open (that is, if it is associated with a file); otherwise, the function returns false.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`return Boolean;`
Returns true if the file handle is open (that is, if it is associated with a file); otherwise, the function returns false.

function Mode

```
function Mode (File : File_Type) return File_Mode;
```

Description

Returns the mode for which the specified file is open.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`return File_Mode;`
Returns the mode for which the specified file is open.

Errors

If the file is not open, the Io_Exceptions.Status_Error exception is raised.

function Name

```
function Name (File : File_Type) return String;
```

Description

Returns the name of the file currently associated with the specified file handle.

For temporary files, this function returns the unique name provided by the Rational Environment during the creation of the file.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`return String;`
Returns the name of the file currently associated with the specified handle.

Errors

If the file is not open, the Io_Exceptions.Status_Error exception is raised.

procedure Open

```
procedure Open (File : in out File_Type;
                Mode :        File_Mode;
                Name :        String;
                Form :        String    := "");
```

Description

Associates the specified file handle with an existing file having the specified name.

Parameters

`File : in out File_Type;`
Specifies the handle for the file.

`Mode : File_Mode;`
Specifies the access mode for which the file is to be used.

`Name : String;`
Specifies the name of the external file to be opened.

`Form : String := "";`
Currently, the Form parameter, if specified, has no effect.

Errors

If the specified file handle is already open, the Io_Exceptions.Status_Error exception is raised.

If the string specified in the Name parameter does not allow the unique identification of a file, the Io_Exceptions.Name_Error exception is raised. In particular, this exception is raised if no file with the specified name exists.

The Io_Exceptions.Use_Error exception is raised under the following conditions:

- The Element_Type type is unsafe (that is, it contains access or task types).
- The file cannot be opened with the specified mode.
- Another job has locked the file.

subtype Positive_Count

```
subtype Positive_Count is Count range 1 .. Count'Last;
```

Description

Defines the valid range of direct file index positions for a nonempty file.

procedure Read

```
procedure Read (File :     File_Type;
                Item : out Element_Type;
                From :     Positive_Count);

procedure Read (File :     File_Type;
                Item : out Element_Type);
```

Description

Reads an item from the specified file and returns the value of this element in the Item parameter.

This procedure operates on a file of the In_File or Inout_File mode. It sets the current index of the specified file to the index value specified by the From parameter. Both forms of the procedure then return, in the Item parameter, the value of the element that resides at the current index. The current index is then increased by 1.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`Item : out Element_Type;`
Specifies the object that receives the value read.

`From : Positive_Count;`
Specifies the position from which the data element is to be read.

Errors

If the file is opened with the Out_File mode, the Io_Exceptions.Mode_Error exception is raised.

If the index exceeds the size of the file, the Io_Exceptions.End_Error exception is raised.

If the element read cannot be interpreted as a value of the Element_Type type, the Io_Exceptions.Data_Error exception is raised.

procedure Reset

```
procedure Reset (File : in out File_Type;
                 Mode :        File_Mode);

procedure Reset (File : in out File_Type);
```

Description

Resets the specified file so that reading from or writing to its elements can be restarted from the beginning of the file.

The file index is set to 1. If a Mode parameter is supplied, the current mode of the given file is set to the specified mode.

Parameters

`File : in out File_Type;`
Specifies the handle for the file.

`Mode : File_Mode;`
Specifies the mode for which the file is to be used when the reset is completed.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

The Io_Exceptions.Use_Error exception is raised under the following conditions:

- The Environment does not support resetting for the file.
- The Environment does not support resetting to the specified mode for the file.
- Another job has locked the file.

procedure Set_Index

```
procedure Set_Index (File : File_Type;
                     To   : Positive_Count);
```

Description

Sets the current index of the file to the specified index value (which may exceed the current size of the file).

This procedure operates on a file of any mode.

Parameters

`File : File_Type;`

Specifies the handle for the file.

`To : Positive_Count;`

Specifies the object to whose value the index is to be set.

function Size

```
function Size (File : File_Type) return Count;
```

Description

Returns the current size of the file associated with the specified file handle.

This function operates on a file of any mode.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`return Count;`
Returns the current size of the file associated with the specified file handle.

procedure Write

```
procedure Write (File : File_Type;
                 Item : Element_Type;
                 To   : Positive_Count);

procedure Write (File : File_Type;
                 Item : Element_Type);
```

Description

Writes the value of the Item parameter to the specified file.

This procedure operates on a file of the Inout_File or Out_File mode. It sets the index of the given file to the index value given by the To parameter. Both forms of the procedure then overwrite the current index of the file with the value of the Item parameter. The current index is then increased by 1.

If a value for the index is greater than the current size of the specified file, the file is automatically extended to include this value.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`Item : Element_Type;`
Specifies the object whose value is to be written.

`To : Positive_Count;`
Specifies the position at which the data element is to be written.

Errors

If the file is opened with the In_File mode, the Io_Exceptions.Mode_Error exception is raised.

If the capacity of the file is exceeded, the Io_Exceptions.Use_Error exception is raised.

end Direct_Io;

package Io_Exceptions

The exceptions in package Io_Exceptions can be raised by I/O operations. The general conditions under which these exceptions can be raised are described in this section. Specific circumstances under which they can be raised are provided for each operation exported by an I/O package. If more than one error condition exists, the corresponding exception that appears earliest in the package is the one that is raised.

Every other I/O package renames one or more of the exceptions exported from this package. Rather than repeat the following descriptions in each of these packages, documentation of the renaming declarations is omitted in the subsequent sections.

The Rational Environment provides additional information about exceptions raised by the I/O packages that describes why a given exception occurred. This information, typically displayed in parentheses after the exception name, is documented in the reference entry for each exception.

exception Data_Error

```
Data_Error : exception;
```

Description

Defines an exception raised by the Read procedure if the element read cannot be interpreted as a value of the required type.

This exception is also raised by a Get or Read procedure if an input sequence fails to satisfy the required syntax or if the value input does not belong to the range of the required type or subtype.

This exception is also raised by the Read and Write procedures of package Polymorphic_Sequential_Io (DIO) if these operations are attempted on files containing unsafe types (that is, containing access or task types as any of their components).

The additional information supplied by the Environment when this exception is raised has the following meaning:

- Input_Syntax_Error: The input value has incorrect syntax.
- Input_Value_Error: The input value is out of range.
- Output_Type_Error: The output value is an unsafe type.
- Output_Value_Error: An attempt has been made to write a value out of range.

exception Device_Error

```
Device_Error : exception;
```

Description

Defines an exception raised if an I/O operation cannot be completed because of a malfunction of the underlying system.

The additional information supplied by the Environment when this exception is raised has the following meaning:

- Device_Data_Error: A hardware error such as a parity error has occurred.
- Illegal_Reference_Error: An illegal reference has been attempted.
- Illegal_Heap_Access_Error: An Illegal_Heap_Access exception was raised when the operation was attempted.
- Page_Nonexistent_Error: A nonexistent page was referenced.
- Write_To_Read_Only_Page_Error: A write to a read-only page was attempted.

exception End_Error

```
End_Error : exception;
```

Description

Defines an exception raised by an attempt to skip (read past) the end of a file.

exception Layout_Error

```
Layout_Error : exception;
```

Description

Defines an exception raised in TIO, packages Text_Io and Io, by a call to operations that violate the limits of Count and by an attempt to put too many characters to a string; also raised in package Window_Io (DIO) by an attempt to position the cursor outside the image boundary.

The additional information supplied by the Environment when this exception is raised has the following meaning:

- Column_Error: A column exceeds the line or page length.
- Illegal_Position_Error: A position parameter is illegal.
- Item_Length_Error: An item length is too big or small.

exception Mode_Error

```
Mode_Error : exception;
```

Description

Defines an exception raised by specifying a file whose mode conflicts with the desired operation.

For example, this exception is raised by a call to Set_Input or Get when a file of the Out_File mode is provided.

The additional information supplied by the Environment when this exception is raised is:

- Illegal_Operation_On_Infile
- Illegal_Operation_On_Outfile

exception Name_Error

```
Name_Error : exception;
```

Description

Defines an exception raised by a call to the Create or Open procedure if the string given for the Name parameter does not allow the identification of a legal unique file.

The Name_Error exception is raised by the Create procedure under any of the following conditions:

- The filename does not conform to the syntax of a name.
- An object of the nonfile class with the same name as the filename already exists in the context in which the creation is attempted.
- The context in which the creation is attempted cannot contain files. Files are allowed only in directories or worlds.

The additional information supplied by the Environment when this exception is raised has the following meaning:

- Ambiguous_Name_Error: A name does not identify a unique object.
- Illformed_Name_Error: A name does not conform to the syntax for a legal Environment filename.
- Nonexistent_Directory_Error: A library in the name does not exist.
- Nonexistent_Object_Error: The specified object does not exist.
- Nonexistent_Version_Error: The specified version of the object does not exist.

exception Status_Error

```
Status_Error : exception;
```

Description

Defines an exception raised by an attempt to operate upon a file handle that is not open and by an attempt to open a file handle that is already open.

The additional information supplied by the Environment when this exception is raised has the following meaning:

- Already_Open_Error: The file handle is already open.
- Not_Open_Error: The file handle is not open.

exception Use_Error

```
Use_Error : exception;
```

Description

Defines an exception raised if an operation is attempted that is not possible for reasons that depend on the file and the executing job's access rights.

This exception is raised by an attempt to create when there are objects of nonfile classes with similar names, by an attempt to open or reset with a mode that is not supported for the file, and by a call to the Open parameter for a terminal object if the terminal is already assigned to a job.

This exception is raised by the Delete procedure, among other circumstances, when the corresponding file is an object that cannot be deleted.

This exception is raised by the Create and Open procedures in packages Direct_Io and Sequential_Io (DIO) if they are attempted with instantiations on unsafe types (that is, types containing access or task types as any of their components).

The additional information supplied by the Environment when this exception is raised has the following meaning:

- Access_Error: There are insufficient access rights to perform the operation.
- Capacity_Error: The output file is full.
- Check_Out_Error: The object is not checked out using the configuration management and version control system.
- Class_Error: There is an existing object of a different class.
- Frozen_Error: An attempt is made to change a frozen object.
- Line_Page_Length_Error: An improper value for line or page length is encountered.
- Lock_Error: Another job has locked the object.
- Reset_Error: The file cannot be reset or have its mode changed.
- Unsupported_Error: The operation is not supported.

end Io_Exceptions;

RATIONAL

package Polymorphic_Sequential_Io

This package provides facilities for sequential I/O upon files whose components are of one or more (nonlimited) types. These types must be safe types, which is any type that does not contain access types or task types as any of its components. If a read or a write operation is attempted with an instantiation on an unsafe type, the Data_Error exception will be raised. This package provides the capabilities similar to those required by the *Reference Manual for the Ada Programming Language*, Chapter 14, for Sequential_Io, except that polymorphism is supported.

The package provides the nested generic package Operations, containing read and write operations, which can be instantiated for each of the desired types.

The fundamental abstraction provided by package Polymorphic_Sequential_Io is the File_Type type. Objects of this type are file handles that can be mapped to files consisting of a sequence of values that are transferred in the order of their appearance.

procedure Append

```
procedure Append (File : in out File_Type;
                 Name :        String;
                 Form :        String     := "");
```

Description

Opens the specified file for writing at the end of the file.

This procedure associates the specified file handle with an existing file having the specified name. The file is left open and the mode is set to Out_File.

Parameters

`File : in out File_Type;`
Specifies the handle for the file.

`Name : String;`
Specifies the name of the external file to be appended.

`Form : String := "";`
Currently, the Form parameter, if specified, has no effect.

Errors

If the file handle is already open, the Io_Exceptions.Status_Error exception is raised.

If the string specified in the Name parameter does not allow the unique identification of a file, the Io_Exceptions.Name_Error exception is raised. In particular, this exception is raised if no file with the specified name exists.

The Io_Exceptions.Use_Error exception is raised when an attempt is made to perform an Append operation on objects on which the Out_File mode is not supported or the file is locked by another job.

procedure Close

```
procedure Close (File : in out File_Type);
```

Description

Severs the association between the specified file handle and its associated file.

Parameters

```
File :  in out File_Type;
```
Specifies the handle for the file.

Errors

If the file is not open, the Io_Exceptions.Status_Error exception is raised.

procedure Create

```
procedure Create (File : in out File_Type;
                 Mode :        File_Mode := Out_File;
                 Name :        String    := "";
                 Form :        String    := "");
```

Description

Establishes a new file with the specified name and associates this file with the specified file handle.

The specified file is left open.

Parameters

`File : in out File_Type;`
Specifies the handle for the file.

`Mode : File_Mode := Out_File;`
Specifies the access mode for which the file is to be used.

`Name : String := "";`
Specifies the name of the file to be created. A null string specifies a file that is not accessible after the completion of the main program (a temporary file).

`Form : String := "";`
Currently, the Form parameter, if specified, has no effect.

Restrictions

Files can be created only in directories or worlds.

Errors

If the file handle is already open, the Io_Exceptions.Status_Error exception is raised.

The Io_Exceptions.Name_Error exception is raised under the following conditions:

- The filename does not conform to the syntax of a name.
- An object of a nonfile class with the same name as the filename already exists in the context in which the creation is attempted.
- The context in which the creation is attempted cannot contain files. Files are allowed only in directories or worlds.

The Io_Exceptions.Use_Error exception is raised under the following conditions:

- The file cannot be opened with the specified mode.
- Another job has locked the file.
- The executing job does not have create access.

---

procedure Delete

```
procedure Delete (File : in out File_Type);
```

Description

Deletes the file associated with the specified file handle.

The file is closed, and the file ceases to exist.

Parameters

`File : in out File_Type;`
Specifies the handle for the file.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

The Io_Exceptions.Use_Error exception is raised under the following conditions:

- The Environment does not support deletion on the file.
- The executing job does not have the access rights required to delete the file.
- Another job has locked the file.

function End_Of_File

```
function End_Of_File (File : File_Type) return Boolean;
```

Description

Returns true if no more elements can be read from the specified file; otherwise, the function returns false.

This function operates on a file of the In_File mode.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`return Boolean;`
Returns true if no more elements can be read from the specified file; otherwise, the function returns false.

Errors

If the file is not open, the Io_Exceptions.Status_Error exception is raised.

If the file is not opened with the In_File mode, the Io_Exceptions.Mode_Error exception is raised.

type File_Mode

```
type File_Mode is (In_File, Out_File);
```

Description

Specifies the mode of access for which a file is open.

In_File denotes a file with read-only access; Out_File denotes a file with write-only access.

type File_Type

```
type File_Type is limited private;
```

Description

Defines a file handle type for files to be processed by operations in this package.

function Form

```
function Form (File : File_Type) return String;
```

Description

Returns the null string ("") in all cases.

When, in the future, the Form parameter to the Create and Open procedures is supported, this function will return the Form value specified in the call to the Create or Open procedure.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`return String;`
Returns the null string ("") in all cases.

Errors

If the file is not open, the Io_Exceptions.Status_Error exception is raised.

References

procedure Create

procedure Open

---

function Is_Open

```
function Is_Open (File : File_Type) return Boolean;
```

Description

Returns true if the file handle is open (that is, if it is associated with a file); otherwise, the function returns false.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`return Boolean;`
Returns true if the file handle is open (that is, if it is associated with a file); otherwise, the function returns false.

function Mode

```
function Mode (File : File_Type) return File_Mode;
```

Description

Returns the mode for which the specified file handle is open.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`return File_Mode;`
Returns the mode for which the specified file handle is open.

Errors

If the file is not open, the Io_Exceptions.Status_Error exception is raised.

---

function Name

```
function Name (File : File_Type) return String;
```

Description

Returns the name of the file currently associated with the file handle.

For temporary files, this function returns the unique name provided by the Rational Environment during the creation of the file.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`return String;`
Returns the name of the file currently associated with the file handle.

Errors

If the file is not open, the Io_Exceptions.Status_Error exception is raised.

procedure Open

```
procedure Open (File : in out File_Type;
                Mode :        File_Mode;
                Name :        String;
                Form :        String     := "");
```

Description

Associates the file handle with an existing file having the specified name and sets the mode of the file to the specified mode.

The specified file is left open.

Parameters

`File : in out File_Type;`
Specifies the handle for the file.

`Mode : File_Mode;`
Specifies the access mode for which the file is to be used.

`Name : String;`
Specifies the name of the file to be opened.

`Form : String := "";`
Currently, the Form parameter, if specified, has no effect.

Errors

If the file handle is already open, the Io_Exceptions.Status_Error exception is raised.

If the string specified in the Name parameter does not allow the unique identification of a file, the Io_Exceptions.Name_Error exception is raised. In particular, this exception is raised if no file with the specified name exists.

The Io_Exceptions.Use_Error exception is raised if the file cannot be opened with the specified mode or if another job has locked the file.

procedure Reset

```
procedure Reset (File : in out File_Type;
                 Mode :        File_Mode);

procedure Reset (File : in out File_Type);
```

Description

Resets the specified file so that reading from or writing to its elements can be restarted from the beginning of the file.

If a Mode parameter is supplied, the current mode of the specified file is set to the specified mode.

Parameters

`File : in out File_Type;`
Specifies the handle for the file.

`Mode : File_Mode;`
Specifies the mode for which the file is to be used when the reset is completed.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

The Io_Exceptions.Use_Error exception is raised under the following conditions:

- The Environment does not support resetting for the file.
- The file cannot be opened with the specified mode.
- Another job has locked the file.

RATIONAL

generic package Operations

The following package exports operations for reading and writing to package Polymorphic_Sequential_Io files. It can be instantiated as many times as required on any safe types to enable objects of these types to be read and to be written to files.

generic formal type Element_Type

```
type Element_Type is private;
```

Description

Denotes the type for which the read and write operations are defined.

Restrictions

If unsafe types (that is, types containing access or task types as any of their components) are used for the instantiation, subsequent read or write operations will raise the Io_Exceptions.Data_Error exception.

procedure Read

```
procedure Read (File :     File_Type;
                Item : out Element_Type);
```

Description

Reads an element from the specified file and returns the value of this element in the Item parameter.

This procedure operates on a file of the In_File mode.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`Item : out Element_Type;`
Specifies the object that receives the value read.

Errors

If the file is not open, the Io_Exceptions.Status_Error exception is raised.

If the file is not opened with the In_File mode, the Io_Exceptions.Mode_Error exception is raised.

If no more elements can be read from the specified file, the Io_Exceptions.End_Error exception is raised.

If the element read cannot be interpreted as a value of the Element_Type type, the Io_Exceptions.Data_Error exception is raised.

If the Element_Type is an unsafe type (that is, it contains access or task types as any of its components), the Io_Exceptions.Data_Error exception is raised.

procedure Write

```
procedure Write (File : File_Type;
                 Item : Element_Type);
```

Description

Writes the value of the Item parameter to the specified file.

This procedure operates on a file of the Out_File mode.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`Item : Element_Type;`
Specifies the object whose value is to be written.

Errors

If the file is not opened with the Out_File mode, the Io_Exceptions.Mode_Error exception is raised.

If the capacity of the file is exceeded, the Io_Exceptions.Use_Error exception is raised.

If the Element_Type is an unsafe type (that is, it contains access or task types as any of its components), the Io_Exceptions.Data_Error exception is raised.

end Operations;

```
end Polymorphic_Sequential_Io;
```

RATIONAL

generic package Sequential_Io

This package provides the capabilities for Sequential_Io as required by the *Reference Manual for the Ada Programming Language*, Chapter 14. It provides facilities for sequential I/O upon files whose components are of the same (nonlimited) type. This type must be a safe type, which is any type that does not contain access types or task types in any of its components. If a create or an open operation is attempted with an instantiation on an unsafe type, the Io_Exceptions.Use_Error exception will be raised.

The fundamental abstraction provided by package Sequential_Io is the File_Type type. Objects of this type are file handles that can be mapped to files consisting of a sequence of values that are transferred in the order of their appearance.

procedure Close

```
procedure Close (File : in out File_Type);
```

Description

Severs the association between the specified file handle and its associated file.

The specified file is left closed.

Parameters

`File : in out File_Type;`
Specifies the handle for the file.

Errors

If the file is not open, the Io_Exceptions.Status_Error exception is raised.

procedure Create

```
procedure Create (File : in out File_Type;
                 Mode :        File_Mode := Out_File;
                 Name :        String    := "";
                 Form :        String    := "");
```

Description

Establishes a new external file with the specified name and associates this file with the specified file handle.

The specified file is left open.

Parameters

`File : in out File_Type;`
Specifies the handle for the file.

`Mode : File_Mode := Out_File;`
Specifies the access mode for which the file is to be used.

`Name : String := "";`
Specifies the name of the file to be created. A null string for the Name parameter specifies a file that is not accessible after the completion of the main program (a temporary file).

`Form : String := "";`
Currently, the Form parameter, if specified, has no effect.

Restrictions

Files can be created only in directories or worlds.

Errors

If the file handle is already open, the Io_Exceptions.Status_Error exception is raised.

The Io_Exceptions.Name_Error exception is raised under the following conditions:

- The filename does not conform to the syntax of a name.
- An object of a nonfile class with the same name as the filename already exists in the context in which the creation is attempted.
- The context in which the creation is attempted cannot contain files. Files are allowed only in directories or worlds.

The Io_Exceptions.Use_Error exception is raised under the following conditions:

- The Element_Type type is unsafe (that is, it contains access or task types).
- The file cannot be opened with the specified mode.
- The executing job does not have create access.
- Another job has locked the file.

---

procedure Delete

```
procedure Delete (File : in out File_Type);
```

Description

Deletes the file associated with the specified file handle.

The specified file is closed, and the file ceases to exist.

Parameters

`File : in out File_Type;`
Specifies the handle for the file.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

The Io_Exceptions.Use_Error exception is raised under the following conditions:

- The Environment does not support deletion on the file.
- The executing job does not have the access rights required to delete the file.
- Another job has locked the file.

generic formal type Element_Type

```
type Element_Type is private;
```

Description

Defines the type of the items that form the files for each particular instantiation.

Restrictions

The type used to instantiate the generic must be a nonlimited type as well as a safe type. Specifically, the type cannot be an access or task type and cannot contain components that are access or task types.

If unsafe types are used for the instantiation, subsequent read or write operations will raise the Io_Exceptions.Data_Error exception.

function End_Of_File

```
function End_Of_File (File : File_Type) return Boolean;
```

Description

Returns true if no more elements can be read from the specified file; otherwise, the function returns false.

This function operates on a file of the In_File mode.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`return Boolean;`
Returns true if no more elements can be read from the specified file; otherwise, the function returns false.

Errors

If the file is not open, the Io_Exceptions.Status_Error exception is raised.

If the file is not opened with the In_File mode, the Io_Exceptions.Mode_Error exception is raised.

type File_Mode

```
type File_Mode is  (In_File, Out_File);
```

Description

Specifies the mode of access for which a file is open.

In_File denotes a file with read-only access; Out_File denotes a file with write-only access.

type File_Type

```
type File_Type is limited private;
```

Description

Defines the file handle type unique to each instantiation of the package.

Objects of this type denote file handles that can be mapped to files.

function Form

```
function Form (File : File_Type) return String;
```

Description

Returns the null string ("") in all cases.

If, in the future, the Form parameter to the Create and Open procedures is supported, this function will return the Form value provided in the call to the Create or Open procedure.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`return String;`
Returns the null string ("") in all cases.

Errors

If the file is not open, the Io_Exceptions.Status_Error exception is raised.

References

procedure Create

procedure Open

function Is_Open

```
function Is_Open (File : File_Type) return Boolean;
```

Description

Returns true if the file handle is open (that is, if it is associated with a file); otherwise, the function returns false.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`return Boolean;`
Returns true if the file handle is open (that is, if it is associated with a file); otherwise, the function returns false.

function Mode

```
function Mode (File : File_Type) return File_Mode;
```

Description

Returns the mode for which the specified file handle is open.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`return File_Mode;`
Returns the mode for which the specified file handle is open.

Errors

If the file is not open, the Io_Exceptions.Status_Error exception is raised.

function Name

```
function Name (File : File_Type) return String;
```

Description

Returns the name of the file currently associated with the specified file handle.

For temporary files, this function returns the unique name provided by the Rational Environment during the creation of the file.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`return String;`
Returns the name of the file currently associated with the specified file handle.

Errors

If the file is not open, the Io_Exceptions.Status_Error exception is raised.

procedure Open

```
procedure Open (File : in out File_Type;
                Mode :        File_Mode;
                Name :        String;
                Form :        String    := "");
```

Description

Associates the file handle with an existing file having the specified name and sets the mode of the file to the specified mode.

The file is left open.

Parameters

`File : in out File_Type;`
Specifies the handle for the file.

`Mode : File_Mode;`
Specifies the access mode for which the file is to be used.

`Name : String;`
Specifies the name of the file to be created.

`Form : String := "";`
Currently, the Form parameter, if specified, has no effect.

Errors

If the file handle is already open, the Io_Exceptions.Status_Error exception is raised.

If the string specified in the Name parameter does not allow the unique identification of a file, the Io_Exceptions.Name_Error exception is raised. In particular, this exception is raised if no file with the specified name exists.

The Io_Exceptions.Use_Error exception is raised under the following conditions:

- The Element_Type type is unsafe (that is, it contains access or task types).
- The file cannot be opened with the specified mode.
- Another job has locked the file.

---

procedure Read

```
procedure Read (File :     File_Type;
                Item : out Element_Type);
```

Description

Reads an element from the specified file and returns the value of this element in the Item parameter.

This procedure operates on a file opened with the In_File mode.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`Item : out Element_Type;`
Specifies the object that receives the value read.

Errors

If the file is not open, the Io_Exceptions.Status_Error exception is raised.

If the file is not opened with the In_File mode, the Io_Exceptions.Mode_Error exception is raised.

If no more elements are available in the file, the Io_Exceptions.End_Error exception is raised.

If the element read cannot be interpreted as a value of the Element_Type type, the Io_Exceptions.Data_Error exception is raised.

procedure Reset

```
procedure Reset (File : in out File_Type;
                 Mode :        File_Mode);

procedure Reset (File : in out File_Type);
```

Description

Resets the specified file so that reading from or writing to its elements can be restarted from the beginning of the file.

If a Mode parameter is supplied, the current mode of the specified file is set to the specified mode.

Parameters

`File : in out File_Type;`
Specifies the handle for the file.

`Mode : File_Mode;`
Specifies the mode for which the file is to be used when the reset is completed.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

The Io_Exceptions.Use_Error exception is raised under any of the following conditions:

- The Environment does not support resetting for the file.
- The file cannot be opened with the specified mode.
- Another job has locked the file.

procedure Write

```
procedure Write (File : File_Type;
                 Item : Element_Type);
```

Description

Writes the value of the Item parameter to the specified file.

This procedure operates on a file of the Out_File mode.

Parameters

`File : File_Type;`
Specifies the handle for the file.

`Item : Element_Type;`
Specifies the object whose value is to be written.

Errors

If the file is not opened with the Out_File mode, the Io_Exceptions.Mode_Error exception is raised.

If the capacity of the file is exceeded, the Io_Exceptions.Use_Error exception is raised.

end Sequential_Io;

package Window_Io

This package provides facilities for performing I/O to windows on the terminal screen.

Package Window_Io offers the user direct control over the creation and manipulation of images contained in windows on the screen. In TIO, packages Text_Io and Io provide simply sequential output to the screen, but package Window_Io allows the user to get and put characters and strings to any place in the image, similar to package Direct_Io. This offers additional flexibility in creating the user interface for tool applications written for the Rational Environment. Whereas Text_Io primarily supports an interrogative style of user interface, Window_Io can be used to create a variety of menu-driven interfaces, form-based input for data, and displays in which the program controls scrolling.

The fundamental abstraction provided by package Window_Io is the File_Type type. Objects of this type (called *file handles* hereafter) essentially denote quarter-plane images. A portion of this image is made visible through a window on the terminal screen. File handles can be opened twice—once for input and once for output. All puts to an image must go through a file handle opened with the Out_File mode. All gets from an image must go through a file handle opened with the In_File mode.

Images are segmented into a matrix of lines and columns. Lines are numbered from 1 starting from the top of the image; columns are numbered from 1 starting from the left side of the image. Each character resides at a particular line and column number. Lines also have length; the length is the column number of the last character on the line, including blank characters. Finally, all images have associated with them the notion of the last line in that image.

Window_Io introduces the notion of a cursor. All images have associated with them a current cursor position. A program can request information about the location of the cursor in an image, and it can also reposition the cursor as required. Input to, and output from, an image is always performed relative to this current cursor position. Input and output operations can also implicitly reposition the cursor during their execution.

Package Window_Io provides access to the terminal's ability to display characters in a variety of fonts and character sets. A program can write read-only text to the screen that users are unable to modify and can write characters designated as a prompt that will disappear when users type on the characters.

An application can also take control of the keystroke stream coming from the terminal keyboard. This control of the keystroke stream allows the program to interpret individual keystrokes and to redefine their resulting effect. Facilities are provided to define mnemonic names for certain keys for more readable use within the program.

Finally, when a program is executing as the current job, applications can take advantage of the Rational Editor commands. For example, an application could use the Commands.Editor.Window.Beginning_Of command (EI) to reposition the window on an image.

Two Case Studies

Package Window_Io offers a powerful set of facilities for creating user interfaces to display information to the user and for requesting information from the user. As in the Rational Editor, information can be displayed in a structured manner and then edited by the user. The edited information can be checked to verify that all user modifications are acceptable and that actions are taken based upon the change. This method offers a much greater degree of flexibility than the sequential interrogative style interfaces available through package Text_Io (TIO).

The objective of this section is to provide a more concrete understanding of the facilities provided by package Window_Io and the application domain to which they apply. The approach will be to discuss in some detail the development of two user interface abstractions, the form and the menu.

Some basic concepts necessary for all window-based applications using package Window_Io will be discussed first. How to set up some useful definitions for display fonts and for names of keyboard keys will be investigated. Then some useful utilities for moving the cursor and manipulating images will be defined.

The first abstraction case study focuses on the essential requirements and implementation strategies for a form as a method of information entry for users. Finally, a variety of menu abstractions for operation selection will be discussed, and one of them will be developed in detail.

Basic Concepts

Images and Windows

All objects in the Environment have an image, part or all of which appears in a window on the terminal screen. Package Window_Io provides facilities for creating and manipulating images in windows. There are two useful ways of thinking about this. In one sense, package Window_Io allows applications to create windows in which text images can be formed. In a more formal sense, Window_Io is another way of creating a text object that has an image displayed in a window. There is one important difference, however. Text objects created by package Window_Io have no corresponding file in the directory structure.

As with other I/O packages, these objects can be opened for input or output, closed and later reopened, and deleted. Consult the reference entries later in this section for the effects of the following procedures:

- procedure Create
- procedure Open
- procedure Close
- procedure Delete

Input to and Output from Images

All images have the notion of a current position for the cursor in that image. All input and output procedures work relative to the current cursor position. Normally, applications will first move the cursor to the appropriate position in an image (unless it is already there) and then call the desired input or output procedure.

Several forms of input are provided for both Character and String types. Basically, an application can either extract a character or string from an image or request that the user provide some input from the keyboard.

Several forms of output are also provided for both Character and String types. An application can either insert text into an image or replace existing text with an Overwrite procedure.

Consult the reference entries in this section for the effects of the following procedures:

- procedure Position_Cursor
- procedure Get
- procedure Get_Line
- procedure Insert
- procedure Overwrite
- procedure New_Line

Operations are also available for deleting text from images, including:

- procedure Delete
- procedure Delete_Lines

Other operations provide information about the image in a window, including:

- procedure Report_Cursor
- function Line_Length
- function Last_Line

Definitions and Utilities

Fonts

All characters are written to the terminal screen in what is called a *font*. Fonts describe exactly how a character will be displayed on the screen. Fonts have two major components: an indication of the character set and an array of display attributes. The Rational Terminal supports two character sets: a normal alphanumeric set and a graphics set. Package Window_Io defines two named constants (Plain and Graphics) for use in defining fonts indicating use of each character set. Some font declarations that applications might find useful are:

```
with Window_Io;
package Fonts is

    Normal       : constant
                   Window_Io.Font := Window_Io.Font'(Window_Io.Plain,
                                                     (others => False));
    Graphics     : constant
                   Window_Io.Font := Window_Io.Font'(Window_Io.Graphics,
                                                     (others => False));

    Inverse_Bold : constant
                   Window_Io.Font := Window_Io.Font'(Window_Io.Plain,
                                                     (Inverse => True,
                                                      Bold => True,
                                                      others => False));
    Underscore   : constant
                   Window_Io.Font := Window_Io.Font'(Window_Io.Plain,
                                                     (Underscore => True,
                                                      others => False));
end Fonts;
```

Notes:

- The Window_Io procedures that display either characters or strings (Insert and Overwrite) include a parameter for indicating the desired font for display.
- These same output procedures also have a parameter for the kind of designation with which characters are to be written. Three kinds of designations are supported:
  - Text: Output is displayed as plain text, which can then be modified by a user with the Rational Editor.
  - Prompt: Output is displayed as a prompt that will disappear when the user types on it. The user can turn a prompt into text with:

    ```
    Commands.Editor.Set.Designation_Off
    ```

  - Protected: Once displayed on the screen, the output is read-only and cannot be modified by the user. If a user does attempt to modify protected text with the Rational Editor, the bell will sound and a message will appear in the Message window indicating that this part of the image is read only.
- Use of the graphics character set for drawing boxes is discussed in "Graphics Utilities," below.

Keys

For applications requiring complete control over keyboard input from the user, package Window_Io offers mechanisms to catch keystrokes, to determine which key was pressed, and to decide on some action as a result.

When the user presses a key on the keyboard, that key is interpreted and placed into the character stream. Normally, keystrokes are passed directly to the Rational Editor, which decides what effect the keystrokes will have. The user has the ability, with package Raw nested within package Window_Io, to interrupt the flow of keystrokes to the editor, to get keys one at a time from the stream, and to initiate an application-specific effect based on the value of the key. Use of this technique is fully described in "Menus," below. The Key type in package Raw is defined as:

```
type Key is new Natural range 0 .. 1023;
```

It is useful to define mnemonic names for keys to enhance the readability of code that requires references to particular keys. Package Key_Names, discussed in the example below, exports some named key objects. In the body of the package, these objects are initialized to an Environment-defined value for a keyboard key during elaboration.

```
with Window_Io;
package Key_Names is

    package Raw renames Window_Io.Raw;
    subtype Key_Type is Raw.Key;

    Up              : Key_Type;
    Down            : Key_Type;
    Left            : Key_Type;
    Right           : Key_Type;

    Window          : Key_Type;
    Window_Up       : Key_Type;
    Window_Down     : Key_Type;
    Definition      : Key_Type;
    Enter           : Key_Type;
    User_Interrupt  : Key_Type;

    Next_Item       : Key_Type;
    Previous_Item   : Key_Type;

    function Is_Alphabet_Key (K : Key_Type) return Boolean;

end Key_Names;

with System_Utilities;
package body Key_Names is

    Is_Found : Boolean;
    Terminal : constant
               String := System_Utilities.Terminal_Type;

begin
    Raw.Value (For_Key_Name => "UP", On_Terminal => Terminal,
               Result => Up, Found => Is_Found);

    Raw.Value (For_Key_Name => "DOWN", On_Terminal => Terminal,
               Result => Down, Found => Is_Found);

    Raw.Value (For_Key_Name => "LEFT", On_Terminal => Terminal,
               Result => Left, Found => Is_Found);

    Raw.Value (For_Key_Name => "RIGHT", On_Terminal => Terminal,
               Result => Right, Found => Is_Found);
    ...
    if Terminal = "VT100" then
        Raw.Value (For_Key_Name => "Numeric_8",
                   On_Terminal => Terminal,
                   Result => Definition, Found => Is_Found);
    elsif Terminal = "Rational"
        Raw.Value (For_Key_Name => "F10", On_Terminal => Terminal,
                   Result => Definition, Found => Is_Found);
    end if;
    ...

end Key_Names;
```

Notes:

- The names of keys for each supported terminal type are located in !Machine.Editor_Data.Visible_Keynames. This information is captured as an enumeration type for each terminal. The string representation of these enumeration values should be passed into the Raw.Value procedure as in the example above.
- The System_Utilities.Terminal_Type function (SMU) returns a string value indicating the type of terminal the user specified at login. Since different terminals may have different mappings from names in Visible_Keynames to actual key values, this function can be used to ensure that key input from a different terminal is interpreted in the same way.
- Since all keyboards do not provide an identical set of keys, each supported terminal will have a different set of enumerated values in Visible_Keynames. In this case, the terminal type will need to be tested before the Raw.Value function is called. Note that, in the body of the package Key_Names example above, the name for Definition is F10 for the Rational Terminal and Numeric_8 for the VT100. This mapping is captured in the Vt100_Commands and Rational_Commands procedures in the !Machine.Editor_Data directory.

Package Raw also exports:

```
subtype Simple_Key is Key range Ø .. 127;
```

Simple keys correspond to ASCII characters so that we can use the ASCII names for references instead of defining our own. The following example demonstrates references to simple keys:

```
    A_Key : Raw.Key;
    ...
begin
    ...
    if A_Key in Raw.Simple_Key then

        case Raw.Convert (A_Key) is

            when 'A' =>
                ...  -- perform some operation indicated by 'A'
            when '?' =>
                ...  -- perform some other operation
            ...
        end case;
    end if;
```

Window Utilities

It is also useful to define and implement utilities for moving the cursor around images and manipulating images in various ways. The following example defines an initial set of utilities (there is certainly an opportunity for several more):

```
with Window_Io;
package Window_Utilities is

    procedure Beginning_Of_Line (Window : Window_Io.File_Type);
    procedure End_Of_Line (Window : Window_Io.File_Type);

    procedure Next_Line (Window : Window_Io.File_Type);
    -- retain the current column position

    procedure Home (Window : Window_Io.File_Type);
    procedure Erase (Window : Window_Io.File_Type);

    procedure Continue (Input_Window : Window_Io.File_Type;
                        Output_Window : Window_Io.File_Type;
                        Prompt : String;
                        Line   : Window_Io.Line_Number;
                        Column : Window_Io.Column_Number);

    function Query (Input_Window : Window_Io.File_Type;
                    Output_Window : Window_Io.File_Type;
                    Prompt : String;
                    Line   : Window_Io.Line_Number;
                    Column : Window_Io.Column_Number) return String;

    type Iterator is private;  -- for all lines in an image

    function Initialize (Window : Window_Io.File_Type) return Iterator;
    function Done (Iter : Iterator) return Boolean;
    function Value (Iter : Iterator) return String;
    procedure Next (Iter : in out Iterator);
private
    type Iterator is ...

end Window_Utilities;
```

Some selected bodies for these utilities are:

```
with Window_Io;
with Fonts;
package body Window_Utilities is

    procedure Next_Line (Window : Window_Io.File_Type) is
        Current_Line   : Window_Io.Line_Number;
        Current_Column : Window_Io.Column_Number;
    begin
        Window_Io.Report_Cursor (Window, Current_Line, Current_Column);

        if Window_Io.Last_Line (Window) /= Current_Line then
            Window_Io.Position_Cursor
                (Window, Current_Line + 1, Current_Column);
        end if;
    end Next_Line;
```

```
    procedure Erase (Window : Window_Io.File_Type) is
    begin
        Window_Io.Position_Cursor (Window);
        Window_Io.Delete_Lines (Window, Window_Io.Last_Line (Window));
    end Erase;

    procedure Continue (Input_Window : Window_Io.File_Type;
                        Output_Window : Window_Io.File_Type;
                        Prompt : String;
                        Line   : Window_Io.Line_Number;
                        Column : Window_Io.Column_Number)
            is separate;

    function Query (Input_Window : Window_Io.File_Type;
                    Output_Window : Window_Io.File_Type;
                    Prompt : String;
                    Line   : Window_Io.Line_Number;
                    Column : Window_Io.Column_Number) return String is

    begin
        Window_Io.Position_Cursor (Output_Window, Line, Column);

        -- write out the prompt
        Window_Io.Overwrite (Output_Window, Prompt,
                             Fonts.Inverse_Bold, Window_Io.Prompt);

        -- reposition the cursor on top of the prompt
        Window_Io.Position_Cursor (Input_Window, Line, Column);

        --  return the user's input
        return Window_Io.Get_Line (Input_Window, "");
    end Query;

end Window_Utilities;
```

The Query procedure may require more explanation. When the cursor is positioned on text written with a prompt designation, the Get_Line procedure waits for the user to input a response. When the user commits the response, the entered characters are returned to the program. To ensure that a prompt is available for the Get_Line procedure, the Query procedure first writes out the prompt and then repositions the cursor on top of it before calling the Get_Line procedure.

The Continue procedure is similar to the Query procedure in that it waits for the user to commit a response. It does not return the response but merely indicates that the user wants to continue. In this case, part of the prompt is written with a protected designation to prevent it from disappearing if the user accidentally types on top of it.

```
separate (Window_Utilities)
procedure Continue (Input_Window  : Window_Io.File_Type;
                    Output_Window : Window_Io.File_Type;
                    Prompt : String;
                    Line   : Window_Io.Line_Number;
                    Column : Window_Io.Column_Number) is

    Char : Character;
begin
    Window_Io.Position_Cursor (Output_Window, Line, Column);

    --insert the prompt to hang on
    Window_Io.Insert (Output_Window, " ",
                      Fonts.Inverse_Bold, Window_Io.Prompt);
    Window_Io.Insert (Output_Window, Prompt,
                      Fonts.Inverse_Bold, Window_Io.Protected);

    Window_Io.Position_Cursor (Input_Window, Line, Column);
    Window_Io.Get (Input_Window, "", Char);

    Window_Io.Position_Cursor (Output_Window, Line, Column);
    Window_Io.Delete_Lines (Output_Window, 2);

end Continue;
```

Graphics Utilities

The Rational Terminal supports a graphics character set that is useful for drawing straight-line structures. A full description of the graphics character set appears in the *Rational Terminal User's Manual.* The following example demonstrates how to draw a box in an image:

```
with Window_Io;
with Fonts;
procedure Draw_Box (Window : Window_Io.File_Type;
                    On_Line : Window_Io.Line_Number;
                    On_Column : Window_Io.Column_Number;
                    Height : Natural;
                    Width  : Natural) is

        Upper_Left_Corner  : constant Character := 'l';
        Upper_Right_Corner : constant Character := 'k';
        Lower_Left_Corner  : constant Character := 'm';
        Lower_Right_Corner : constant Character := 'j';
        Vertical_Line      : constant Character := 'x';
        Horizontal_Line    : constant Character := 'q';

    begin
        Window_Io.Position_Cursor (Window, On_Line, On_Column);
        Window_Io.Overwrite (Window, Upper_Left_Corner, Fonts.Graphics);

        for I in 1 .. Width loop
            Window_Io.Overwrite
                    (Window, Horizontal_Line, Fonts.Graphics);
        end loop;
```

```
        Window_Io.Overwrite (Window, Upper_Right_Corner, Fonts.Graphics);

        Window_Io.Position_Cursor (Window, On_Line + 1, On_Column);

        for I in 1 .. Height loop
            Window_Io.Overwrite (Window, Vertical_Line, Graphics);
            Window_Io.Move_Cursor (Window, 1, - 1);
        end loop;

        Window_Io.Position_Cursor
                  (Window, On_Line + 1, On_Column + Width + 1);

        for I in 1 .. Height loop
            Window_Io.Overwrite (Window, Vertical_Line, Graphics_Set);
            Window_Io.Move_Cursor (Window, 1, - 1);
        end loop;

        Window_Io.Position_Cursor
                  (Window, On_Line + Height + 1, On_Column);
        Window_Io.Overwrite (Window, Lower_Left_Corner, Graphics_Set);

        for I in 1 .. Width loop
            Window_Io.Overwrite (Window, Horizontal_Line, Graphics_Set);
        end loop;

        Window_Io.Overwrite (Window, Lower_Right_Corner, Graphics_Set);
    end Draw_Box;
```

The Form Abstraction

A *form* provides a method of getting structured information from a user in a somewhat unstructured way. A form with various entries defined by the application can be displayed in a window. Control should then be returned to the Rational Editor to allow completion of the form by the user. With the full power of the editor available to the user, information can be entered in any order using any of the editor features. When the user indicates that the form is complete, the image should be parsed and responses returned to the application for interpretation.

Other considerations might include the ability to indicate errors in user responses, to redisplay the form for correction with the editor, and then to resubmit the response.

In particular, it might be desirable for the display to look something like this:

NAME : *[Input]*

ADDRESS : *[Input]*

TELEPHONE NUMBER : *[(Area Code) Number]*

AGE : *[Positive Number]*

One possible specification for this abstraction might be the following:

```
with Fonts;
with Window_Io;
with Unbounded_String;
generic
    type Form_Item is (<>);  --Defines entries of the form
    with function Image (Item : Form_Item) return String;
    -- provides a string representing the name of the entry
package Forms is

    type Form_Entry is private;
    -- defines a prompt and display attributes for an entry

    type Form_Definition is array (Form_Item) of Form_Entry;

    procedure Initialize (Definition : in out Form_Definition);
    -- initialize all entries with default values

    procedure Modify (The_Entry : in out Form_Entry;
                      New_Prompt : String;
                      Font : Window_Io.Font := Fonts.Inverse_Bold;
                      Kind : Window_Io.Designation := Window_Io.Prompt);
    -- allows modification of an entry's prompt

    procedure Display (Output_Window : Window_Io.File_Type;
                       Form : Form_Definition);

    procedure Get_User_Response
                  (Form_Output : Window_Io.File_Type;
                   Form_Input : Window_Io.File_Type;
                   Definition : in out Form_Definition);

    function Response (An_Entry : Form_Entry) return String;
    -- multiple line input is separated by Ascii.Lf characters

    Unable_To_Parse_Response : exception;

private
    Item_Size : constant := 80;
    package Unbounded is new Unbounded_String (Item_Size);

    type Form_Entry is
        record
            Prompt      : Unbounded.Variable_String;
            Prompt_Font : Window_Io.Font;
            Prompt_Kind : Window_Io.Designation;
        end record;

end Forms;
```

Design Issues

- This particular choice of specification is generic on a discrete type and is intended to be instantiated with an enumeration type. The image function is used by the Display procedure to write out entry names. If this spec were instantiated with the following type:

```
type Form_Entries is (Name, Address, Telephone_Number, Age);
```

  the image of the form presented above could be accomplished.

- The Modify procedure is intended to allow an application to redefine its own prompt strings or to modify the prompt display attributes to indicate errors.
- Note that the Get_User_Response procedure contains an ***in out*** Form_Definition parameter. This allows the image to be parsed and the responses to be returned in the Form_Definition itself.
- It is expected that an application using this abstraction would generally call the Display procedure before calling Get_User_Response. The display operation might have been used internally in Get_User_Response and might not have been exported. If, however, an application desires to display a form with errors and asks the user whether or not to continue editing, the display must be separate; it has been made so for this reason.
- The Response function can be used to return the user's response for a particular entry.

Implementation Issues

See the body of package Forms in the following example for reference.

- The most interesting issue is how to return control to the Rational Editor to allow the user to complete the form. This is accomplished by positioning the cursor at the end of the image and calling the Window_Io.Get procedure with the null string for the prompt. Since the cursor is at the end of the file, the program will wait until the user commits the response. The value returned from the Get procedure is not important and is not looked at. The Get procedure is used only to signal that the user has completed editing and that the form can now be parsed.
- If the prompt string passed to the Modify procedure is the null string, the prompt string is unchanged, but the fonts and designation are changed. This is useful for retaining the user's response and modifying the display attributes to indicate an error.

```
with Window_Utilities;
with String_Utilities;
package body Forms is
    procedure Modify (The_Entry : in out Form_Entry;
                      New_Prompt : String;
                      Font : Window_Io.Font := Fonts.Inverse_Bold;
                      Kind : Window_Io.Designation := Window_Io.Prompt) is
    begin
        if New_Prompt /= "" then
            Unbounded.Copy (The_Entry.Prompt, New_Prompt);
        end if;
        The_Entry.Prompt_Font := Font;
        The_Entry.Prompt_Kind := Kind;
    end Modify;

    procedure Display (Output_Window : Window_Io.File_Type;
                       Form : Form_Definition) is
        A_Char              : Character;
        First_Prompt_Line   : Window_Io.Line_Number;
        First_Prompt_Column : Window_Io.Column_Number;
        First_Prompt_Found  : Boolean := False;
    begin
        Window_Utilities.Erase (Output_Window);
        Window_Io.Position_Cursor (Output_Window);

        for Item in Form_Item loop
            Window_Io.Insert
                 (Output_Window, Image (Item) & " : ",
                  Kind => Window_Io.Protected);

            for I in 1 .. Unbounded.Length (Form (Item).Prompt) loop
                A_Char := Unbounded.Char_At (Form (Item).Prompt, I);

                if A_Char = Ascii.Lf then
                    Window_Io.New_Line (Output_Window, 1);
                else
                    Window_Io.Insert
                         (Output_Window, A_Char,
                          Image => Form (Item).Prompt_Font,
                          Kind => Form (Item).Prompt_Kind);
                end if;
            end loop;
            Window_Io.New_Line (Output_Window, 1);
        end loop;
    end Display;

    procedure Parse (Input_Window : Window_Io.File_Type;
                     Form : in out Form_Definition) is separate;
    -- implementation of a body for parse is left to the user

    procedure Get_User_Response (Form_Output : Window_Io.File_Type;
                                 Form_Input : Window_Io.File_Type;
                                 Definition : in out Form_Definition) is
        Out_Char : Character;
    begin
        Window_Io.Get (Form_Input, "", Out_Char);
        Parse (Form_Input, Definition);
    end Get_User_Response;
end Forms;
```

The Menu Abstraction

The *menu* abstraction offers a rich set of user interface options for applications. Generally, it presents a set of selections to the user that, when activated, produces some effect. Other kinds of menus are collections of objects whose images are displayed by the menu. Objects can be selected and operations applied to them. This may or may not change their image in the menu.

The menu in the example below can be used by an application to offer a set of choices to a user. Selection of a choice implies that some operation will be performed. The user should be able to move from choice to choice with the arrow keys and to indicate selection with the Enter key. Selections might also be made by pressing the first letter of the menu choice's name.

An application will require a way to build menu definitions, display them, and cause an operation to be performed when a particular menu choice is selected. Another issue is the layout of the menu on the screen. The two options that this example will offer are a vertical layout and a horizontal layout.

The following example is a generic specification for the simple menu described above:

```
with Window_Io;
with New_Keys;
generic
    type Element is private;

    with function Line_Image (E : Element) return String;

    with procedure Apply_Operation (To_Element : Element;
                                    Window : Window_Io.File_Type;
                                    Column_Offset : Natural := 0;
                                    Line_Offset : Natural := 0);

    with function Is_Quit_Key return Boolean;
    with function Is_Selection_Key return Boolean;

package Single_Selection_Electric_Menus is
    subtype Window_Type is Window_Io.File_Type;

    type Menu_Definition is private;

    function Make return Menu_Definition;
    procedure Add (E : Element; To : in out Menu_Definition);

    type Layout is (Vertical, Horizontal);

    procedure Get_User_Selection
                  (Menu : Window_Type; Definition : Menu_Definition;
                   Column_Offset : Natural := 0;
                   Line_Offset : Natural := 0;
                   Presentation : Layout := Vertical);
```

```
private
    type Node;
    type Menu_Definition is access Node;
    type Node is
        record
            Elem        : Element;
            First_Char  : Character;
            Line        : Window_Io.Line_Number;
            Column      : Window_Io.Column_Number;
            Next        : Menu_Definition;
            Previous    : Menu_Definition;
        end record;
end Single_Selection_Electric_Menus;
```

Design Issues

The following discussion refers to the example in the previous section, "The Menu Abstraction."

- This example is generic on the Element type for the menu selection. The Element type must have a line image and an Apply_Operation procedure that will initiate some operation based on the value of the element. Note that this procedure has some additional parameters to indicate the current window and some offsets within that window. These parameters are essential if the operation one wants to perform is the display of another nested menu.
- Generic formals are functions identifying which keys indicate that the user would like to quit the menu and which keys indicate the selection of a particular operation. The arrow keys for traversal over the elements of the menu have been hard-wired into the body but could have been made generic as well. The layout form (either horizontal or vertical) might also affect which arrow keys are operative. A horizontal layout might use the left and right arrows; a vertical layout might use the up and down arrow keys.
- The package exports a Make operation that builds an empty menu definition. The user can then add elements to build up the final definition.
- No Display procedure is exported. There seems to be no reason to separate the display from requesting a response from the user. Since the user could manipulate the display image, in some situations, before being asked for a response, it seems unwise to offer the opportunity for no reason. If a compelling reason were identified, the Display procedure from the body could be exported easily.
- Another option for the Get_User_Selection operation is to make it a function that returns the element selected. The client program will then determine the operation to be performed. The penalty is that the client program will have to decide when to quit. In this case it would also be necessary to export the display operation to avoid having to redisplay the menu each time a user selection is requested.

Implementation Issues

Global issues will be discussed in this section. Issues pertinent to individual subprograms will be addressed with the corresponding code.

- A doubly linked list has been chosen to implement menu definitions. Note that each node of the list also stores information about the position of the element in the display and about the first character of the element's image for electric completion.
- Several helper functions have been defined in the body:
  - Find_Def: Returns the next node of the menu definition whose element begins with the character First. If an element is not found, the Definition_Not_Found exception is raised.
  - Initialize_Placement: Determines the layout of all elements in the menu.
  - Display: Writes the images for each element of the menu at the appropriate position.
  - Erase: Erases the menu image from the image. Note that, when nested menus are displayed in the same window, only the current menu should be erased.

```
package body Single_Selection_Electric_Menus  is

    Definition_Not_Found  : exception;

    function Make return Menu_Definition is
    begin
        return null;
    end Make;

    procedure Add (E : Element; To : in out Menu_Definition)  is
        Temp       : Menu_Definition;
        Line : constant String := Line_Image (E);
    begin
        if To = null then
            To           := new Node'(E, Line (Line'First),
                                      1, 1, null, null);
            To.Next      := To;
            To.Previous  := To;
        else
            Temp           := To.Previous;
            Temp.Next      := new Node'(E, Line (Line'First),
                                        1, 1, null, Temp);
            Temp.Next.Next  := To;
            To.Previous     := Temp.Next;
        end if;
    end Add;
    -- helper functions for Get_User_Response
    function Find_Def
            (First : Character; Def : Menu_Definition)
                     return Menu_Definition is separate;
```

```
    procedure Initialize_Placement
              (Def : in Menu_Definition;
               Column_Offset : Natural; Line_Offset : Natural;
               Presentation : Layout) is separate;

    procedure Display
              (Menu : Window_Type; Definition : Menu_Definition;
               Column_Offset : Natural; Line_Offset : Natural;
               Presentation : Layout) is separate;

    procedure Erase
              (Menu : Window_Type; Definition : Menu_Definition;
               Column_Offset : Natural := 0) is separate;

    procedure Get_User_Selection
              (Menu : Window_Type; Definition : Menu_Definition;
               Column_Offset : Natural := 0;
               Line_Offset : Natural := 0;
               Presentation : Layout := Vertical) is separate;

end Single_Selection_Electric_Menus;

separate (Single_Selection_Electric_Menus)
function Find_Def (First : Character;
                   Def : Menu_Definition) return Menu_Definition is
   Temp : Menu_Definition := Def;
begin
    if Temp = null then
        raise Definition_Not_Found;
    elsif Temp.First_Char = First then
        return Temp;
    else
        Temp := Temp.Next;

        while Temp /= Def loop
            if Temp.First_Char = First then
                return Temp;
            else
                Temp := Temp.Next;
            end if;
        end loop;
    end if;
    raise Definition_Not_Found;
end Find_Def;

separate (Single_Selection_Electric_Menus)
procedure Initialize_Placement
              (Def : in Menu_Definition;
               Column_Offset : Natural; Line_Offset : Natural;
               Presentation : Layout) is

    Temp        : Menu_Definition := Def;
    Next_Line   : Positive := Line_Offset + 1;
    Next_Column : Positive := Column_Offset + 1;
begin
    Temp.Line   := Next_Line;
    Temp.Column := Next_Column;
```

```
    case Presentation is
        when Vertical =>
            Next_Line := Next_Line + 1;
        when Horizontal =>
            Next_Column := Next_Column +
                Line_Image (Temp.Elem)'Length + 4;

            if Next_Column > 80 then
                Next_Column := Column_Offset + 1;
                Next_Line := Next_Line + 1;
            end if;
    end case;

    Temp := Temp.Next;

    while Temp /= Def loop
        Temp.Line   := Next_Line;
        Temp.Column := Next_Column;

        case Presentation is
            when Vertical =>
                Next_Line := Next_Line + 1;
            when Horizontal =>
                Next_Column := Next_Column +
                    Line_Image (Temp.Elem)'Length + 4;
                if Next_Column > 80 then
                    Next_Column := Column_Offset + 1;
                    Next_Line := Next_Line + 1;
                end if;
        end case;
        Temp := Temp.Next;
    end loop;
end Initialize_Placement;

with Fonts;
separate (Single_Selection_Electric_Menus)
procedure Display
          (Menu : Window_Type; Definition : Menu_Definition;
           Column_Offset : Natural; Line_Offset : Natural;
           Presentation : Layout) is

    Temp_Def : Menu_Definition := Definition;
begin

    Window_Io.Position_Cursor
         (Menu, Temp_Def.Line, Temp_Def.Column);
    Window_Io.Overwrite (Menu, Line_Image (Temp_Def.Elem),
                         Fonts.Inverse_Bold);
    Temp_Def := Temp_Def.Next;

    while Temp_Def /= Definition loop
        Window_Io.Position_Cursor
             (Menu, Temp_Def.Line, Temp_Def.Column);
        Window_Io.Overwrite (Menu, Line_Image (Temp_Def.Elem),
                             Fonts.Normal);

        Temp_Def := Temp_Def.Next;
    end loop;
```

```
    -- repositions the cursor on the first element
    Window_Io.Position_Cursor
         (Menu, Temp_Def.Line, Temp_Def.Column);

end Display;
```

Most of the interesting issues are associated with the next example, an implementation of the Get_User_Selection procedure. In this procedure, the menu is first displayed in the window. The raw keystroke stream is then opened and One_Key is taken from the stream. The value of this key is then interpreted. If the key is a traversal key (an arrow key), the currently highlighted element is unhighlighted and the new node is highlighted. If the key is a letter of the alphabet, an attempt is made to find an element that begins with that character; the Apply_Operation procedure is called if an element is found. The Apply_Operation procedure is also called when the key indication user selection is pressed. If the quit key is pressed, the menu raw stream is closed, the menu is erased, and the Get_User_Selection procedure is exited. If any other key is pressed, the terminal bell is sounded.

Note that the raw stream is closed before each call to Apply_Operation. It is then reopened after the call returns. This example establishes this convention for good reason. Package Raw does not export an operation to indicate whether or not the stream is open. Thus, a program cannot tell whether it has to perform an Open procedure. If a program tries to open a stream that is already open, the Io_Exceptions.Status_Error exception is raised. The convention of a closed character stream across procedure calls was established for this implementation.

```
separate (Single_Selection_Electric_Menus)
procedure Get_User_Selection
          (Menu : Window_Type; Definition : Menu_Definition;
           Column_Offset : Natural := 0;
           Line_Offset : Natural := 0;
           Presentation : Layout := Vertical) is

    Current_Node   : Menu_Definition := Definition;
    Selected_Font  : Window_Io.Font := Fonts.Inverse_Bold;

    package Raw renames Window_Io.Raw;
    Character_Stream  : Raw.Stream_Type;
    One_Key           : Raw.Key;
    function "=" (A, B : Raw.Key) return Boolean renames Raw."=";
begin

    Initialize_Placement (Definition, Column_Offset, Line_Offset,
                          Presentation);
    Display (Menu, Definition, Column_Offset, Line_Offset,
                          Presentation);

    Raw.Open (Character_Stream);
    loop
        Window_Io.Position_Cursor
             (Menu, Current_Node.Line, Current_Node.Column);

        Raw.Get (Character_Stream, One_Key);

        -- interpretation of the keystroke from the user:
```

```
if (Presentation = Vertical and One_Key = New_Keys.Up) or
   (Presentation = Horizontal and One_Key = New_Keys.Left)
  then
    Window_Io.Position_Cursor
         (Menu, Current_Node.Line, Current_Node.Column);
    Window_Io.Overwrite        -- turn off selection
         (Menu, Line_Image (Current_Node.Elem), Fonts.Normal);

    Current_Node := Current_Node.Previous;

    Window_Io.Position_Cursor
         (Menu, Current_Node.Line, Current_Node.Column);
    Window_Io.Overwrite       -- turn on selection
         (Menu, Line_Image (Current_Node.Elem), Selected_Font);


elsif (Presentation = Vertical and One_Key = New_Keys.Down) or
      (Presentation = Horizontal and One_Key = New_Keys.Right)
   then
    Window_Io.Position_Cursor
         (Menu, Current_Node.Line, Current_Node.Column);
    Window_Io.Overwrite        -- turn off selection
         (Menu, Line_Image (Current_Node.Elem), Fonts.Normal);

    Current_Node := Current_Node.Next;

    Window_Io.Position_Cursor
         (Menu, Current_Node.Line, Current_Node.Column);
    Window_Io.Overwrite        -- turn on selection
         (Menu, Line_Image (Current_Node.Elem), Selected_Font);

-- electric selection on first character:
elsif New_Keys.Is_Alphabet_Key (One_Key) then
    declare
        Char     : Character :=
            String_Utilities.Upper_Case
             (Window_Io.Raw.Convert (One_Key));
        New_Node : Menu_Definition;
    begin
        New_Node := Find_Def (Char, Current_Node.Next);

        Window_Io.Position_Cursor
             (Menu, Current_Node.Line, Current_Node.Column);
        Window_Io.Overwrite
             (Menu, Line_Image (Current_Node.Elem), Fonts.Normal);

        Current_Node := New_Node;

        Window_Io.Position_Cursor
             (Menu, Current_Node.Line, Current_Node.Column);
        Window_Io.Overwrite (Menu,
                  Line_Image (Current_Node.Elem), Selected_Font);

        Raw.Close (Character_Stream);
        Apply_Operation (Current_Node.Elem, Menu,
                         Column_Offset, Line_Offset);

        -- ensure correct cursor position after apply
        Window_Io.Position_Cursor
```

```
                    (Menu, Current_Node.Line, Current_Node.Column);

                Raw.Open (Character_Stream);
            exception
                when Definition_Not_Found =>
                    Window_Io.Bell (Menu);
            end;

        elsif Is_Select_Key (One_Key) then

            Raw.Close (Character_Stream);
            Apply_Operation (Current_Node.Elem, Menu,
                             Column_Offset, Line_Offset);
            Window_Io.Position_Cursor
                 (Menu, Current_Node.Line, Current_Node.Column);
            Raw.Open (Character_Stream);

        elsif Is_Quit_Key (One_Key) then
            Raw.Close (Character_Stream);
            Erase (Menu, Definition, Column_Offset);
            exit;

        else
            Window_Io.Bell (Menu);
        end if;
    end loop;
end Get_User_Selection;
```

Disconnecting from a Menu

One useful operation for applications that capture the key stream is to allow the user to leave the current window to do something else, possibly returning later to continue working. An application might decide to recognize window traversal operations such as Window.Up and Window.Down. The question now becomes how to disconnect from the current job and wait for the user to indicate a desire to reconnect. It is desirable to allow this without forcing the user to interrupt the program and then explicitly reconnect to the numbered job using the Job.Connect procedure.

The solution to this is fairly straightforward. When the program recognizes that the user intends to leave (a [Window] - [↑] key sequence, for example), the program disconnects itself and then waits on a prompt with a call to Window_Io.Get. This is exactly the same approach used to return the user to the Rational Editor in a form. When the user terminates the Get procedure with a Commit procedure, the job is implicitly reconnected, continuing to process keystrokes from the user. Since the user is returned to the Rational Editor and has the ability to modify the image in window, it is a good idea to redraw the image, if possible, to ensure the integrity of the display. The following program fragment should provide the basic implementation approach.

```
procedure Hang (Output_Window : Window_Io.File_Type;
                Input_Window : Window_Io.File_Type;
                Key : Raw.Key) is

begin
    Window_Utilities.Home (Output_Window);
    Window_Io.New_Line (Output_Window, 1);

    if Key = New_Keys.Up then
        Editor.Window.Previous;
    elsif Key = New_Keys.Down then
        Editor.Window.Next;
    else
        null;
    end if;

    Job.Disconnect;
    -- note that the program moved to the new window with the
    -- editor before disconnecting; calls to the editor can
    -- come only from connected jobs

   Window_Utilities.Continue
         (Input_Window, Output_Window,
          Prompt => "Type ENTER on this window to Reconnect",
          Line => 1, Column => 1);
end Hang;
```

The following example is a fragment of the key processor that recognizes the user's intent to leave and call the Hang procedure. It also calls a procedure to reset the window image when the user reconnects.

```
    ...
    elsif One_Key = New_Keys.Window_Up then
        Hang (Menu_Output, Menu_Input, New_Keys.Up);
        Reset_Screen;
    elsif One_Key = New_Keys.Window_Down then
        Hang (Menu_Output, Menu_Input, New_Keys.Down);
        Reset_Screen;
    elsif One_Key = New_Keys.Window then
        Raw.Get (Character_Stream, Second_Key);

        if Second_Key = New_Keys.Up then
            Hang (Menu_Output, Menu_Input, New_Keys.Up);
            Reset_Screen;
        elsif Second_Key = New_Keys.Down then
            Hang (Menu_Output, Menu_Input, New_Keys.Down);
            Reset_Screen;
        else
            Window_Io.Bell (Menu_Output);
        end if;
    ...
```

type Attribute

```
type Attribute is
    record
        Bold        : Boolean;
        Faint       : Boolean;
        Underscore  : Boolean;
        Inverse     : Boolean;
        Slow_Blink  : Boolean;
        Rapid_Blink : Boolean;
        Unused_0    : Boolean;
        Unused_1    : Boolean;
    end record;
```

Description

Defines the attributes that characters can have when displayed on the screen.

A character's display depends on the user's terminal setup. The actual effect of the Inverse attribute depends on the background mode currently in use. The Bold and Faint attributes indicate display with brighter green if the terminal is set up in dim mode, and they will display dimmer green if the terminal is set up to display in bold.

The Rational Terminal and the terminal controller do not support all possible combinations of the attribute fields. The available combinations anticipate the most useful combinations. In general, the attribute fields have the following effects (restrictions are described below):

Bold	Character appears in brighter green
Faint	Character appears in brighter green
Underscore	Character is underlined
Inverse	Character background appears in the inverse of the terminal's current background
Slow_Blink	Character blinks
Rapid_Blink	Character blinks
Unused_0	Reserved
Unused_1	Reserved

Restrictions

Not all combinations are supported.

The terminal supports only two brightness levels. In the normal setting (when all attribute fields are false), characters are written at the normal brightness level. Setting either the Bold or the Faint attribute to true writes characters in the bold font for the terminal.

Setting the Underscore, the Inverse, and either the Bold or the Faint attributes to true at the same time is not supported. In this case, characters are displayed in bold and inverse but not underscored.

Only one blink speed is currently supported; setting either the Slow_Blink or the Rapid_Blink attribute field to true makes characters displayed with these attributes blink.

When either blink attribute field is set, the following combinations are currently supported:

Otherwise plain	Bold, Faint, Inverse, and Underscore set to false.
Inverse only	Bold, Faint, and Underscore set to false; Underscore sets to true.
Underline only	Bold, Faint, and Inverse set to false; Underscore sets to true.
Inverse bold	Underscore sets to false: Inverse and either Bold or Faint set to true.

Example

```
Inverse_Bold_Attribute : constant Window_Io.Attribute :=
            (Bold => True, Inverse => True, others => False);
```

References

Rational Terminal User's Manual

---

procedure Bell

```
procedure Bell (File : File_Type);
```

Description

Rings the terminal bell.

Note: There is only one bell. It can be rung with any Window_Io file handle, even those handles that have not been opened.

Parameters

`File : File_Type;`

Specifies the handle for an image. Since there is only one bell, any handle can be used, even those handles that have not been opened.

type Character_Set

```
type Character_Set is new Natural range 0 .. 15;
```

Description

Defines the possible character sets for the display.

Restrictions

Currently only two character sets are supported. Plain (0) indicates the standard alphanumeric character set. Graphics (1) indicates the graphics character set supported by the terminal.

Currently the graphics character set is displayable only with the plain or blinking attributes.

References

Rational Terminal User's Manual

function Char_At

```
function Char_At (File : File_Type) return Character;
```

Description

Returns the character at the current cursor position.

Parameters

`File : File_Type;`
Specifies the handle for the image containing the character in question.

`return Character;`
Returns the character at the current cursor position.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

procedure Close

```
procedure Close (File : in out File_Type);
```

Description

Removes access to the image with this file handle.

The image is not deleted or removed from the terminal screen.

Parameters

`File : in out File_Type;`
Specifies the handle for the image.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

subtype Column_Number

```
subtype Column_Number is Positive;
```

Description

Defines the column number of a character in an image.

Columns are numbered starting with 1 from the far left side of the image.

subtype Count

```
subtype Count is Natural;
```

Description

Defines the number of times that an operation should be repeated.

procedure Create

```
procedure Create (File : in out File_Type;
                 Mode :        File_Mode := Out_File;
                 Name :        String;
                 Form :        String    := "");
```

Description

Creates an image for performing I/O.

Normally, a new empty image is created when this procedure is called, and a window containing the image appears on the terminal screen. All named images can be opened twice, once for input and once for output. If an image is already open for the current job with the specified name and is open with a mode other than the one currently requested, the existing image will be opened for the new mode.

Parameters

`File : in out File_Type;`
Specifies the file handle for the created image.

`Mode : File_Mode := Out_File;`
Specifies the access mode for which the image is to be used.

`Name : String;`
Specifies the name of the image to be created. This name appears on the left side of the banner of the window containing the image.

`Form : String := "";`
Currently, the Form parameter, if specified, has no effect.

Errors

If the named image is already open for the designation mode, the Io_Exceptions.Status_Error exception is raised.

function Default_Font

```
function Default_Font (For_Type : Designation) return Font;
```

Description

Returns the default font for each kind of designation.

For both the text and protected designations, the attributes are all Vanilla (that is all set to false). The prompt designation returns a font whose Inverse attribute is set to true. All designations use the Plain character set in their default font.

Parameters

`For_Type : Designation;`

Specifies a particular designation.

`return Font;`

Returns default settings for the character set and attributes.

Example

This function can be used with one of the output procedures to indicate the desired font:

```
    Window_Image : Window_Io.File_Type;
    Text_Designation : Window_Io.Designation := Window_Io.Text;
    ...
begin
    Window_Io.Open
      (Window_Image, Window_Io.Out_File, "Banner Name");

    Window_Io.Insert (File  => Window_Image,
                      Item  => "Some String",
                      Image => Window_Io.Default_Font
                                  (Text_Designation),
                      Kind  => Text_Designation);
```

References

type Attribute

type Designation

type Font

constant Vanilla

---

procedure Delete

```
procedure Delete (File : in out File_Type);
```

Description

Deletes the image associated with the file handle and removes the window containing the image from the terminal screen and the Window Directory.

Any other handles associated with this image are implicitly closed.

Parameters

`File : in out File_Type;`
Specifies the file handle for the image to be deleted.

Errors

If the image associated with the file handle has already been deleted, the Io_Exceptions.Status_Error exception is raised.

procedure Delete

```
procedure Delete (File       : File_Type;
                  Characters : Count);
```

Description

Deletes the specified number of characters from the current line, starting with the character at the current cursor position.

The position of the cursor is unchanged. If the count specified is greater than the number of characters remaining on the line, all the subsequent characters on that line are deleted and no exception is raised.

Parameters

`File : File_Type;`
Specifies the file handle for the image.

`Characters : Count;`
Specifies the number of characters to be deleted.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

If the file handle is not open for output (with the Out_File mode), the Io_Exceptions-.Mode_Error exception is raised.

procedure Delete_Lines

```
procedure Delete_Lines (File  : File_Type;
                        Lines : Count      := 1);
```

Description

Deletes the specified number of lines from the image, starting at the current line.

The column number of the cursor is unchanged, but it will be placed on the line following the last deleted line. If the count specified is greater than the number of lines remaining in the image, all the subsequent lines in that image are deleted and no exception is raised.

Parameters

`File : File_Type;`
Specifies the file handle for the image.

`Lines : Count := 1;`
Specifies the number of lines to be deleted.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

If the file handle is not open for output (with the Out_File mode), the Io_Exceptions-.Mode_Error exception is raised.

type Designation

```
type Designation is (Text, Prompt, Protected);
```

Description

Defines the behavior of edited characters and strings written to an image.

Enumerations

Prompt

Displays output as a prompt that disappears when the user types on it. The user can turn a prompt into text with Commands.Editor.Set.Designation_Off (EI).

Protected

Displays output that is protected (that is, read-only output, which cannot be modified by the user).

Text

Displays output as plain text that can be modified by a user with the Rational Editor.

Restrictions

Users can delete protected fields by using the Rational Editor to enclose a field completely inside a region; the entire region, including the protected field, can then be deleted.

Example

The designation for an output operation is specified as the Kind parameter to the Insert procedure:

```
    Window_Image : Window_Io.File_Type;
    ...
begin
    Window_Io.Open
      (Window_Image, Window_Io.Out_File, "Banner Name");

    Window_Io.Insert (File  => Window_Image,
                      Item  => "Test program script",
                      Image => Window_Io.Default_Font
                                    (Window_Io.Text);
                      Kind  => Window_Io.Text);

    Window_Io.Insert (File  => Window_Image,
                      Item  => "Enter a file name: ",
                      Image => Window_Io.Default_Font
                                    (Window_Io.Protected);
                      Kind  => Window_Io.Protected);

    Window_Io.Insert (File  => Window_Image,
                      Item  => "Name of a file",
                      Image => Window_Io.Default_Font
                                    (Window_Io.Prompt);
                      Kind  => Window_Io.Prompt);
```

function End_Of_File

```
function End_Of_File (File : File_Type) return Boolean;
```

Description

Returns true if the cursor is positioned at the end of the last line in the image; otherwise, the function returns false.

Parameters

`File : File_Type;`
Specifies the file handle for the image in question.

`return Boolean;`
Returns true if the cursor is positioned at the end of the last line in the image; otherwise, the function returns false.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

References

procedure Position_Cursor

function End_Of_Line

```
function End_Of_Line (File : File_Type) return Boolean;
```

Description

Returns true if the cursor is positioned at the end of the line; otherwise, the function returns false.

Parameters

`File : File_Type;`
Specifies the file handle for the image containing the line in question.

`return Boolean;`
Returns true if the cursor is positioned at the end of the line; otherwise, the function returns false.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

References

procedure Position_Cursor

type File_Mode

```
type File_Mode is (In_File, Out_File);
```

Description

Defines the access mode for which an image can be opened.

An image can be opened for one or both modes with a separate file handle.

Enumerations

`In_File`

Denotes an image with read-only access.

`Out_File`

Denotes an image with write-only access.

Example

```
    Output_Window : Window_Io.File_Type;
    Input_Window : Window_Io.File_Type;
begin
    Window_Io.Open
            (Output_Window, Window_Io.Out_File, "Banner Name");
    Window_Io.Open
            (Input_Window, Window_Io.In_File, "Banner Name");
```

type File_Type

```
type File_Type is private;
```

Description

Defines a handle for an image.

type Font

```
type Font is
    record
        Kind : Character_Set;
        Look : Attribute;
    end record;
```

Description

Defines the way in which ASCII characters written to an image are displayed.

Example

Note the named constant declaration for Normal:

```
Normal : constant Font := Font'(Plain, Vanilla);
```

References

constant Normal

constant Plain

constant Vanilla

function Font_At

```
function Font_At (File : File_Type) return Font;
```

Description

Returns the font of the character that appears at the current cursor position.

If the cursor is positioned after the last character on a line, the line is padded at the end with blanks written with the Normal font. Thus, the Font_At function returns Normal in this case.

Parameters

`File : File_Type;`
Specifies the handle for the image containing the character in question.

`return Font;`
Returns the font of the character that appears at the current cursor position.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

function Form

```
function Form (File : File_Type) return String;
```

Description

Returns the null string ("") regardless of the value provided to a call to the Open or the Create procedure.

In the future, the Form parameter will be supported, and this function will return the actual value provided to the Open or the Create procedure.

Parameters

`File : File_Type;`
Specifies the handle of the file in question.

`return String;`
Returns the null string ("") regardless of the value provided to a call to the Open or the Create procedure.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

procedure Get

```
procedure Get (File   :     File_Type;
               Prompt :     String      := "[input]";
               Item   : out Character);
```

Description

Returns the character at the current cursor position in an image.

This procedure has three slightly different effects depending on the cursor position and the designation of the character written at that position:

- If the character located at the current cursor position is not written with a prompt designation (that is, with either a text or a protected designation), that character is immediately returned in the Item parameter. The cursor is repositioned after the character extracted from the image. The actual image remains unchanged.
- If the character at the current cursor position has been written with a prompt designation, execution is suspended until the user provides the requested character reply.
- If the cursor is located at the end of the image, the prompt string is displayed and execution is suspended until the user provides the requested character reply.

Note that in the second and third cases above, the program does not return until the user commits the response.

Parameters

`File : File_Type;`
Specifies the file handle for the image.

`Prompt : String := "[input]";`
Specifies the string for use as a prompt when querying the user.

`Item : out Character;`
Specifies the object in which to place the requested character.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

If the file handle is not open for input (with the In_File mode), the Io_Exceptions-.Mode_Error exception is raised.

Example 1

In the image:

```
1234567890
ABCDEFGHIJ
```

in which all characters are written with the text designation, the numerals are located on line 1 of the image, and the cursor is positioned at line 2, column 7, the character returned in the Item parameter of the Get procedure is G. The new cursor position is at line 2, column 8.

Example 2

In the image:

```
1234567890 PROMPT STRING
```

if PROMPT STRING is written as a prompt and the cursor is positioned at line 2, column 7, a call to the Get procedure suspends waiting for user input. When the user types a character, the whole prompt string disappears and is replaced with the entered character. If the user then commits the input, the entered character is returned in the Item parameter.

References

type Designation

procedure Get

function Get_Line

---

procedure Get

```
procedure Get (File   :     File_Type;
               Prompt :     String    := "[input]";
               Item   : out String);
```

Description

Returns string input from the image.

This procedure has three slightly different effects depending on the cursor position and the designation of the character written at that position:

- If the character located at the current cursor position is not written with a prompt designation (that is, with either a text or a protected designation), the sequence of characters starting at the current cursor position and continuing for the length of the Item variable is returned in the Item parameter. The cursor is repositioned after the last character read from the image. The actual image remains unchanged.
- If the character at the current cursor position has been written with a prompt designation, execution is suspended until the user provides the number of characters necessary to fill the Item parameter.
- If the cursor is located at the end of the image, the prompt string is displayed and execution is suspended until the user provides the requested number of characters.

Notes:

- In the second and third cases above, the program does not return until the user commits the response.
- If the user intends to use this procedure to extract a string from an image but specifies an item string that has more characters than remain in the image, the user will be prompted with the specified prompt string at the end of the image for the remainder of the characters necessary to fill the Item string completely.

Parameters

```
File : File_Type;
```

Specifies the handle for the image.

```
Prompt : String := "[input]";
```

Specifies the string for use as a prompt when querying the user.

```
Item : out String;
```

Specifies the object receiving the input from the image.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

If the file handle is not open for input (with the In_File mode), the Io_Exceptions-.Mode_Error exception is raised.

Example 1

In the image:

```
1234567890
ABCDEFGHIJ
abcdefghij
```

in which the numerals are located on line 1 of the image and the cursor is positioned at line 2, column 7, the following program fragment returns the string GHI in the Extract parameter with no suspension of execution:

```
    An_Image : Window_Io.File_Type;
    ...
    Extract : String (1 .. 3);
    ...
begin
    ...
    Window_Io.Get (An_Image, "Prompt string", Extract);
```

If Extract were instead declared as:

```
Extract : String (1 .. 10);
```

the returned string would be GHIJ*abcde, in which the * character is actually an Ascii.Lf character.

Finally, if Extract were instead declared as:

```
Extract : String (1 .. 20);
```

the first 16 characters of the returned string would be filled with GHIJ*abcdefghij*. The Prompt string prompt would then be displayed on the next line, and the first four characters of the user's input would fill out the four characters necessary to complete the string.

Example 2

The following procedure can be used to query the user for input at any point in an image. The program first inserts a prompt at the desired position, then repositions the cursor onto the prompt, and finally calls the Get procedure to retrieve the user's response. This corresponds to the second case in the description above.

```
procedure Query (Input_Window     : Window_Io.File_Type;
                 Output_Window    : Window_Io.File_Type;
                 The_Prompt       : String;
                 Line_Position    : Positive;
                 Column_Position  : Positive;
                 Reply : out String) is
begin
  Window_Io.Position_Cursor (Output_Window, Line_Position,
                             Column_Position);

  -- write out the prompt
  Window_Io.Overwrite (Output_Window, The_Prompt,
                       Window_Io.Default_Font
                                 (Window_Io.Prompt),
                       Window_Io.Prompt);

  -- reposition the cursor on top of the prompt
  Window_Io.Position_Cursor (Output_Window, Line_Position,
                             Column_Position);

  -- request user input
  Window_Io.Get (Input_Window, "", Reply);

end Query;
```

References

type Designation

---

function Get_Line

```
function Get_Line (File   : File_Type;
                   Prompt : String    := "[input]") return String;
```

Description

Returns string input from a line in an image.

This procedure has three slightly different effects depending on the cursor position and the designation of the character written at that position:

- If the character located at the current cursor position is not written with a prompt designation (that is, with either a text or a protected designation), the string returned will contain characters starting with the character located at the current cursor position through the last character on that line. Use of this procedure has the side effect of repositioning the cursor at the first character of the next line. The actual image remains unchanged, however.
- If the character at the current cursor position has been written with a prompt designation, execution will be suspended until the user provides the requested string.
- If the cursor is located at the end of the image, the prompt string is displayed and execution will be suspended until the user provides the requested string.

Notes:

- Extraction of an entire line is best accomplished with the Line_Image function also declared within this package.
- The program will not return until the user commits the response.
- A fringe case occurs when the cursor is located on the last line of the image. In this case, the Get_Line function partially fills the return string with the characters in the line but also prompts the user at the end of the line for more characters. Any characters offered by the user are appended to the return string.

Parameters

`File : File_Type;`
Specifies the handle for the image.

`Prompt : String := "[input]";`
Specifies the string for use as a prompt when querying the user.

```
return String;
```

Returns string input from a line in an image.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

If the file handle is not open for input (with the In_File mode), the Io_Exceptions-.Mode_Error exception is raised.

Example 1

In the image:

```
1234567890
ABCDEFGHIJ
abcdefghij
```

in which the numerals are located on line 1 of the image and the cursor is positioned at line 2, column 7, the Get_Line function returns the string `GHIJ`.

Example 2

This example provides a functional form of querying the user for input. The program first inserts a prompt at the desired position, then repositions the cursor onto the prompt, and finally calls the Get_Line procedure to retrieve the user's response.

```
function Query (Input_Window    : Window_Io.File_Type;
                Output_Window   : Window_Io.File_Type;
                The_Prompt      : String;
                Line_Position   : Positive;
                Column_Position : Positive) return String is
begin
  Window_Io.Position_Cursor (Output_Window, Line_Position,
                             Column_Position);

  -- write out the prompt
  Window_Io.Overwrite (Output_Window, The_Prompt,
                       Window_Io.Default_Font
                                 (Window_Io.Prompt),
                       Window_Io.Prompt);

  -- reposition the cursor on top of the prompt
  Window_Io.Position_Cursor (Output_Window, Line_Position,
                             Column_Position);

  -- request user input
  return Window_Io.Get_Line (Input_Window, "");

end Query;
```

References

type Designation

function Line_Image

---

procedure Get_Line

```
procedure Get_Line (File   :     File_Type;
                    Prompt :     String    := "[input]";
                    Item   : out String;
                    Last   : out Natural);
```

Description

Returns string input from a line in an image.

This procedure has three slightly different effects depending on the cursor position and the designation of the character written at that position:

- If the character located at the current cursor position is not written with a prompt designation (that is, with either a text or a protected designation), the string returned in the Item parameter will contain characters starting with the character located at the current cursor position through the last character on that line. The Last parameter indicates the index of the last *valid* character index of the Item parameter. If the length of the Item string is not large enough to hold all remaining characters on that line, the returned string will contain only the subset that will fit. Use of this procedure has the side effect of repositioning the cursor after the last character of the string extracted from the image. The actual image remains unchanged, however.
- If the character at the current cursor position has been written with a prompt designation, execution is suspended until the user provides the requested string. The actual characters returned in the Item parameter are governed by the rules outlined in the first item above.
- If the cursor is located at the end of the image, the prompt string is displayed and execution will be suspended until the user provides the requested string.

Notes:

- The program will not return until the user commits the response.
- The user should not depend on the validity of any characters after the index indicated by the value of the Last parameter.
- If the cursor is positioned after the last character on a line, the Last parameter is set to 0, indicating that no valid output characters were placed in the Item parameter.
- A fringe case occurs when the cursor is located on the last line of the image. In this case, the Get_Line procedure partially fills the return string with the characters in the line but also prompts the user at the end of the line for more characters. Any characters offered by the user are appended to the Item string until it is full.

Parameters

```
File : File_Type;
```
Specifies the handle for the image.

```
Prompt : String := "[input]";
```
Specifies the string for use as a prompt when querying the user.

```
Item : out String;
```
Specifies the container for the input from the image.

```
Last : out Natural;
```
Specifies the index of the last character read into the Item string.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

If the file handle is not open for input (with the In_File mode), the Io_Exceptions-.Mode_Error exception is raised.

Example

In the image:

```
1234567890
ABCDEFGHIJ
abcdefghij
```

in which the numerals are located on line 1 of the image and the cursor is positioned at line 2, column 7, the following program fragment returns the string GHI in the Extract parameter and the Last parameter equals 3 with no suspension of execution.

```
    An_Image : Window_Io.File_Type;
    ...
    Extract : String (1 .. 3);
    Last : Natural;
    ...
begin
    ...
    Window_Io.Get_Line (An_image, "Prompt string", Extract, Last);
```

If Extract were instead declared as:

```
Extract : String (1 .. 1Ø);
```

the returned string would be `GHIJ??????` with the Last parameter equal to **4**. The `?` character means that its value cannot be depended upon.

References

type Designation

---

constant Graphics

```
Graphics : constant Character_Set := 1;
```

Description

Defines a named constant for the graphics character set.

A complete description of the graphics character set is provided in the *Rational Terminal User's Manual.*

procedure Insert

```
procedure Insert (File  : File_Type;
                  Item  : Character;
                  Image : Font        := Normal;
                  Kind  : Designation := Text);

procedure Insert (File  : File_Type;
                  Item  : String;
                  Image : Font        := Normal;
                  Kind  : Designation := Text);
```

Description

Inserts a character or string into the current line at the current cursor position.

The character at the current cursor position and all subsequent characters on that line are shifted to the right. If the cursor is positioned beyond the last character on a line, the Insert procedure will place the character or string in the image beginning at the current cursor position and will fill the intervening space with blanks. In every case, the actual cursor position is positioned after the last character in the inserted string.

Note: To prevent unnecessary scrolling, the screen cursor (the actual cursor on the screen) is not placed at the position of the image cursor but remains at the original position before the insert. Multiple inserts will still work off the image cursor position, placing the inserted characters in the image. The screen cursor can be resynchronized with the actual cursor with a call to the Move_Cursor or the Position_Cursor procedure.

Parameters

`File : File_Type;`
Specifies the handle for the image.

`Item : Character;`
Specifies the character to be inserted.

`Item : String;`
Specifies the string to be inserted.

`Image : Font := Normal;`
Specifies the desired font for display.

```
Kind : Designation := Text;
```

Specifies the desired designation of the display.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

If the file handle is not open for output (with the Out_File mode), the Io_Exceptions-.Mode_Error exception is raised.

References

procedure Move_Cursor

procedure Position_Cursor

---

function Is_Open

```
function Is_Open (File : File_Type) return Boolean;
```

Description

Returns true if the file handle is open; otherwise, the function returns false.

Parameters

`File : File_Type;`
Specifies the handle for the image in question.

`return Boolean;`
Returns true if the file handle is open; otherwise, the function returns false.

function Job_Number

```
function Job_Number return String;
```

Description

Returns a string representing the predefined field name Job_Number for the banner of a window.

This is useful for input to the Set_Banner procedure and the Read_Banner function.

Example

```
Window_Io.Read_Banner (A_Window, Window_Io.Job_Number);
```

returns the job number from the banner of `A_Window`.

References

function Read_Banner

procedure Set_Banner

function Job_Time

```
function Job_Time return String;
```

Description

Returns a string representing the predefined field name Job_Time for the banner of a window.

This is useful for input to the Set_Banner procedure and the Read_Banner function.

Example

```
Window_Io.Read_Banner  (A_Window, Window_Io.Job_Time);
```

returns the job time from the banner of `A_Window`.

References

function Read_Banner

procedure Set_Banner

function Last_Line

```
function Last_Line (File : File_Type) return Line_Number;
```

Description

Returns the number of the last line in the image.

If the image is empty (that is, contains no characters), the line number returned from the Last_Line function is 1.

Parameters

`File : File_Type;`
Specifies the handle for the image in question.

`return Line_Number;`
Returns the number of the last line in the image.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

Example

One easy way to iterate through all the lines in an image is:

```
for Line_Number in 1 .. Window_Io.Last_Line (An_Image) loop
...  -- perform some operation on line Line_Number
end loop;
```

function Line_Image

```
function Line_Image (File : File_Type) return String;
```

Description

Returns the image of the line on which the cursor currently resides.

The string returned includes all characters in the line including trailing blanks but not including a line terminator character. This function has higher performance than the Get_Line procedure and is generally preferred for extracting text from an image.

Parameters

`File : File_Type;`
Specifies the handle for the image in question.

`return String;;`
Returns the image of the line on which the cursor currently resides.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

Example

One method of examining all lines in an image is:

```
for Line_Number in 1 .. Window_Io.Last_Line (An_Image) loop
    Examine (Window_Io.Line_Image (Line_Number));
end loop;
```

function Line_Length

```
function Line_Length (File : File_Type) return Count;
```

Description

Returns the number of characters in the line in which the cursor currently resides.

Parameters

`File : File_Type;`
Specifies the handle for the image in question.

`return Count;`
Returns the number of characters in the current line.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

subtype Line_Number

```
subtype Line_Number is Positive;
```

Description

Defines the legal range for line numbers in an image.

function Mode

```
function Mode (File : File_Type) return File_Mode;
```

Description

Returns the mode for which the specified file handle has been opened.

Parameters

`File : File_Type;`
Specifies the handle for the image in question.

`return File_Mode;`
Returns the mode for which the specified file handle has been opened.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

procedure Move_Cursor

```
procedure Move_Cursor (File          : File_Type;
                       Delta_Lines   : Integer;
                       Delta_Columns : Integer;
                       Offset        : Natural   := 0);
```

Description

Repositions the cursor relative to its current position in the image.

Parameters

`File : File_Type;`
Specifies the handle for the image.

`Delta_Lines : Integer;`
Specifies the number of lines to move the cursor. The cursor is moved down for positive numbers and up for negative numbers.

`Delta_Columns : Integer;`
Specifies the number of columns to move the cursor. The cursor is moved to the right for positive numbers and to the left for negative numbers.

`Offset : Natural := 0;`
Specifies the position of the window relative to the cursor position. With a positive offset, the top of the window is placed that number of lines above the new position of the cursor. With an offset of 0, the cursor is made visible in the window using the normal editing defaults.

Restrictions

If the number specified by the offset would place the cursor outside the window, the window is positioned using normal editing defaults.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

If either the resulting line number or the column number of the new cursor position is less than 1, the Io_Exceptions.Layout_Error exception is raised.

function Name

```
function Name (File : File_Type) return String;
```

Description

Returns the name of the image that was specified when the file handle was created or opened.

Parameters

`File : File_Type;`
Specifies the handle of the image in question.

`return String;`
Returns the name of the image that was specified when the file handle was created or opened.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

procedure New_Line

```
procedure New_Line (File  : File_Type;
                    Lines : Count      := 1);
```

Description

Inserts the specified number of lines after the current line.

If the cursor is positioned in the middle of a line of characters, a line terminator is inserted, effectively breaking the line into two lines.

The cursor is positioned at the beginning of the line following the last inserted line.

Parameters

`File : File_Type;`
Specifies the handle for the image.

`Lines : Count := 1;`
Specifies the number of lines to be inserted.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

If the file handle is not open for output (with the Out_File mode), the Io_Exceptions.Mode_Error exception is raised.

Example

This procedure is most commonly used after a call to the Insert procedure to terminate the line:

```
Window_Io.Insert (An_Image, "Some text forming a line");
Window_Io.New_Line (An_Image, 1);
```

constant Normal

```
Normal : constant Font := Font'(Plain, Vanilla);
```

Description

Defines a named constant for a font, selecting the normal settings for the character set and attributes.

References

constant Plain

constant Vanilla

procedure Open

```
procedure Open (File : in out File_Type;
                Mode :        File_Mode := Out_File;
                Name :        String;
                Form :        String    := "");
```

Description

Opens a file handle for input or output with its corresponding image.

Images can be opened twice, once for input and once for output.

If no image has been created previously with the specified name, a new image is created and a window containing the image will appear on the terminal screen. If an image with the specified name has been created previously but has been closed, the old image can be reopened. If an image is currently open with the specified name and mode, a new image is created and displayed on the terminal screen.

Parameters

`File : in out File_Type;`
Specifies the handle for the opened image.

`Mode : File_Mode := Out_File;`
Specifies the access mode for which the image is to be used.

`Name : String;`
Specifies the name of the image to be created. This name will appear on the left side of the banner of the window containing the image.

`Form : String := "";`
Currently, the Form parameter, if specified, has no effect.

Example

Commonly, images are opened both for input and output:

```
    Input_Window  : Window_Io.File_Type;
    Output_Window : Window_Io.File_Type;

begin
   Window_Io.Open (Input_Window,  Window_Io.In_File,  "WINDOW IO");
   Window_Io.Open (Output_Window, Window_Io.Out_File, "WINDOW IO");
```

Note the use of two file handle objects, one for each mode.

References

procedure Create

procedure Overwrite

```
procedure Overwrite (File  : File_Type;
                     Item  : Character;
                     Image : Font        := Normal;
                     Kind  : Designation := Text);

procedure Overwrite (File  : File_Type;
                     Item  : String;
                     Image : Font        := Normal;
                     Kind  : Designation := Text);
```

Description

Replaces characters or strings in the current line beginning at the current cursor position.

If the new string contains more characters than exist on the current line, the line is extended to include all characters in the new string. If the cursor is positioned beyond the last character on a line, the Overwrite procedure places the character or string in the image beginning at the current cursor position and fills the intervening space with blanks. In every case, the actual cursor is positioned in the image after the last character in the overwritten string.

Note: To prevent unnecessary scrolling, the screen cursor (the actual cursor on the screen) is not placed at the position of the image cursor but remains at the original position before the overwrite. Multiple overwrites will still work off the image cursor, placing the overwritten string in the image. The screen cursor can be resynchronized with the actual cursor with a call to the Move_Cursor or the Position_Cursor procedure.

Parameters

`File : File_Type;`

Specifies the handle for the image.

`Item : Character;`

Specifies the character over which to write the existing character.

`Item : String;`

Specifies the string over which to write the existing string.

```
Image : Font := Normal;
```

Specifies the desired font for display.

```
Kind : Designation := Text;
```

Specifies the desired designation of the display.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

If the file handle is not open for output (with the Out_File mode), the Io_Exceptions-.Mode_Error exception is raised.

References

procedure Move_Cursor

procedure Position_Cursor

---

constant Plain

```
Plain : constant Character_Set := Ø;
```

Description

Defines a named constant for the alphanumeric character set.

procedure Position_Cursor

```
procedure Position_Cursor (File   : File_Type;
                           Line   : Line_Number   := Line_Number'First;
                           Column : Column_Number := Column_Number'First;
                           Offset : Natural       := 0);
```

Description

Places the cursor at the specified line and column.

If the new cursor position is beyond the last character on a line, the length of the line is extended up to the new cursor position and the intervening space is filled with blanks.

If the new cursor position is beyond the currently defined last line in the image, the number of the last line in the image is updated to reflect the new cursor position.

If the new cursor position is outside the current window, the window is repositioned relative to the new cursor position, either through specification of a positive offset (defined below) or through the use of the default offset, with an orientation selected by the Rational Editor.

Parameters

`File : File_Type;`
Specifies the handle for the desired image.

`Line : Line_Number := Line_Number'First;`
Specifies the line on which the cursor should be positioned.

`Column : Column_Number := Column_Number'First;`
Specifies the column on which the cursor should be positioned.

`Offset : Natural := 0;`
Specifies the position of the window relative to the cursor position. With a positive offset, the top of the window is placed the specified number of lines above the new position of the cursor. With an offset of 0, the cursor is made visible in the window using the normal editing defaults.

Restrictions

If the number specified by the offset would place the cursor outside the window, the window is positioned using normal editing defaults.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

---

subtype Positive_Count

```
subtype Positive_Count is Count range 1 .. Count'Last;
```

Description

Defines the allowable range for cursor positions.

function Read_Banner

```
function Read_Banner (File       : File_Type;
                      Field_Name : String) return String;
```

Description

Returns the text residing in the specified field of the banner of the window.

Parameters

`File : File_Type;`

Specifies the handle for the image in question.

`Field_Name : String;`

Specifies the desired field name. Field names are of the form Field_0, Field_1, ..., Field_9. All other values are ignored.

Field_0	Reserved.
Field_1	Corresponds to the job number.
Field_2	Corresponds to the start time of the job.
Field_3..9	Available to the user.

Currently, the fields for the job number and the job time also can be selected with the corresponding Job_Number and Job_Time functions.

`return String;`

Returns the text residing in the specified field of the banner of the window.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

function Read_Banner
package !Io.Window_Io

References

function Job_Number

function Job_Time

procedure Set_Banner

---

procedure Report_Cursor

```
procedure Report_Cursor (File   :     File_Type;
                         Line   : out Line_Number;
                         Column : out Column_Number);
```

Description

Identifies the current position of the cursor in the image.

Parameters

`File : File_Type;`
Specifies the handle of the image in question.

`Line : out Line_Number;`
Specifies the number of the line on which the cursor resides.

`Column : out Column_Number;`
Specifies the number of the column on which the cursor resides.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

Example

This procedure is often useful for writing simple positioning utilities:

```
with Window_Io;
procedure End_Of_Line (Window : Window_Io.File_Type) is

    Current_Line   : Window_Io.Line_Number;
    Current_Column : Window_Io.Column_Number;

begin
    Window_Io.Report_Cursor (Window, Current_Line,
                                     Current_Column);
    Window_Io.Position_Cursor (Window, Current_Line,
                               Window_Io.Line_Length (Window));
end End_Of_Line;
```

procedure Report_Location

```
procedure Report_Location (File   :     File_Type;
                           Line   : out Line_Number;
                           Column : out Column_Number);
```

Description

Reports the location on the terminal screen of the upper-left border of the window containing the specified image.

The upper-left corner of the terminal screen is line 1, column 1.

Parameters

`File : File_Type;`

Specifies the handle for the image.

`Line : out Line_Number;`

Specifies the number of the line on the terminal screen on which the upper-left corner of the window resides.

`Column : out Column_Number;`

Specifies the number of the column on the terminal screen on which the upper-left corner of the window resides.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

procedure Report_Origin

```
procedure Report_Origin (File   :     File_Type;
                         Line   : out Line_Number;
                         Column : out Column_Number);
```

Description

Reports the location of the upper-left corner of the window in the specified image.

Parameters

`File : File_Type;`
Specifies the handle of the image.

`Line : out Line_Number;`
Specifies the line number of the upper-left corner of the window in the specified image.

`Column : out Column_Number;`
Specifies the column number of the upper-left corner of the window in the specified image.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

procedure Report_Size

```
procedure Report_Size (File    :     File_Type;
                       Lines   : out Positive_Count;
                       Columns : out Positive_Count);
```

Description

Reports the number of lines and columns in a window.

Essentially, this procedure reports the amount of space available in a window. The size of the image is unrelated to this data. The number of lines in an image is reported by the Last_Line function.

Parameters

`File : File_Type;`
Specifies the handle of the window in question.

`Lines : out Positive_Count;`
Specifies the number of lines in the window.

`Columns : out Positive_Count;`
Specifies the number of columns in the window.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

Example

This procedure can be useful for ensuring that the window will never scroll, which may occur if characters are written outside the available size of the window:

```
with Window_Io;
procedure Check_Insert (Window : Window_Io.File_Type;
                        Item   : String;
                        Image  : Window_Io.Font;
                        Kind   : Window_Io.Designation) is

    Number_Of_Lines      : Window_Io.Line_Number;
    Number_Of_Columns    : Window_Io.Column_Number;

    Current_Line   : Window_Io.Line_Number;
    Current_Column : Window_Io.Column_Number;

begin
    Window_Io.Report_Cursor (Window, Current_Lines,
                                     Current_Columns);
    Window_Io.Report_Size   (Window, Number_Of_Lines,
                                     Number_Of_Columns);

    if Current_Column + Item'Size > Number_Of_Lines then
        raise Constraint_Error;
    else
        Window_Io.Insert (Window, Item, Image, Kind);
    end if;

end End_Of_Line;
```

References

function Last_Line

procedure Set_Banner

```
procedure Set_Banner (File       : File_Type;
                      Field_Name : String;
                      Value      : String);
```

Description

Substitutes a new image for a particular field name in the banner of a window.

Parameters

`File : File_Type;`
Specifies the handle for the window.

`Field_Name : String;`
Specifies the desired field name. Field names are of the form Field_0, Field_1, ..., Field_9. All other values are ignored.

Field_0	Reserved.
Field_1	Corresponds to the job number.
Field_2	Corresponds to the start time of the job.
Field_3..9	Available to the user.

Currently, the fields for the job number and the job time also can be selected with the corresponding Job_Number and Job_Time functions.

`Value : String;`
Specifies the new field image.

Errors

If the file handle is not open, the Io_Exceptions.Status_Error exception is raised.

Example

```
Set_Banner (An_Image, "Field_3", "Some string");
```

References

function Job_Number

function Job_Time

constant Vanilla

```
Vanilla : constant Attribute := (others => False);
```

Description

Defines a named constant for all attribute fields set to false.

package Raw

This package allows programs to capture raw input from the terminal keyboard.

Keystrokes can be considered the basic unit of data from the user. Instead of sending keystroke input to the Rational Editor, a program can capture keystrokes directly and interpret them as desired. When keystrokes are taken from the raw terminal stream, they are not automatically echoed to the terminal. This allows an application using these facilities to be extremely flexible in its response to input from the user.

There is only one raw keystroke stream per terminal port. Streams are not available on a per-window basis. Only connected jobs can take input from the keyboard. When a job is disconnected, keystroke input is redirected to the Rational Editor.

procedure Close

```
procedure Close (Stream              : in out Stream_Type;
                 Flush_Pending_Input :        Boolean     := False);
```

Description

Disables the program's access to keystrokes from the keyboard and, if requested, discards any remaining characters in the buffer.

Subsequent keystrokes are directed to the Rational Editor.

Parameters

`Stream : in out Stream_Type;`

Specifies the handle for the keyboard character stream.

`Flush_Pending_Input : Boolean := False;`

Specifies whether any remaining characters currently in the stream should be purged from the stream. If Flush_Pending_Input is false, a subsequent attempt to open the stream would find any pending input characters still available in the stream. If Flush_Pending_Input is true, any additional characters originally in the stream would not be available on a subsequent attempt to open the stream.

Errors

If the specified stream is not currently open, the Io_Exceptions.Status_Error exception is raised.

Example

Because there is no way to tell whether or not the character stream is open, it is a good idea to maintain a convention of always keeping the stream either opened or closed across calls to other subprograms. This will ensure that keystrokes are not requested from a closed stream or that an attempt is made to reopen an already opened stream.

```
    Character_Stream : Raw.Stream_Type
    One_Key : Raw.Key;
    ...
begin

    Raw.Open (Character_Stream);

    loop
        Raw.Get (Character_Stream, One_Key);

        if One_Key = ... then

            Raw.Close (Character_Stream);

            Process (One_Key);

            Raw.Open (Character_Stream);
        ...
        else
            Raw.Close (Character_Stream);
            exit;
        end if;
     end loop;
```

function Convert

```
function Convert (C : Character) return Simple_Key;
```

Description

Returns the corresponding simple key for all characters.

Parameters

`C : Character;`
Specifies the character in question.

`return Simple_Key;`
Returns the simple key for the specified character.

function Convert

```
function Convert (K : Simple_Key) return Character;
```

Description

Returns the corresponding character for a given simple key.

Parameters

`K : Simple_Key;`
Specifies the simple key in question.

`return Character;`
Returns the character corresponding to the specified simple key.

Example

This function is often used to find the corresponding character for a simple key:

```
    A_Key : Raw.Key;
...
begin
...
    if A_Key in Raw.Simple_Key then
      case Raw.Convert (A_Key) is
          when 'A' => ...;
      ...
      end case;
```

procedure Disconnect

```
procedure Disconnect (Stream : in out Stream_Type);
```

Description

Frees the user's keyboard and returns input to the Rational Editor.

The stream remains open, allowing the job to wait for input if the user decides to reconnect to the job.

Parameters

`Stream : in out Stream_Type;`
Specifies the handle for the stream.

procedure Get

```
procedure Get (Stream :     Stream_Type;
               Item   : out Key);

procedure Get (Stream :     Stream_Type;
               Item   : out Key_String);
```

Description

Retrieves a key or series of keys from the stream.

If there are no pending keys in the stream, or not enough keys to fill the desired string, execution of the program is suspended until the user enters the required number of keystrokes at the keyboard.

Parameters

`Stream : Stream_Type;`
Specifies the handle for the stream.

`Item : out Key;`
Specifies the requested key.

`Item : out Key_String;`
Specifies the requested series of keys.

Errors

If the handle for the stream is not open, the Io_Exceptions.Status_Error exception is raised.

Example

One method of using this procedure is:

```
    A_Key : Raw.Key;
    Quit_Key : constant Raw.Key := ...;

    The_Stream : Raw.Stream_Type;
...
begin
    Raw.Open (The_Stream);
    loop
        Raw.Get (The_Stream, A_Key);
        if A_Key = ... then
            ...
        elsif A_Key = ... then
            ...          elsif A_Key = Quit_Key then
            Raw.Close (The_Stream);
            exit;
        else
            ...
    end loop;
```

function Image

```
function Image (For_Key     : Key;
                On_Terminal : Terminal) return String;
```

Description

Returns the corresponding image for a key on a particular terminal type as defined in the Environment package !Machine.Editor_Data.Visible_Keynames.

Parameters

`For_Key : Key;`
Specifies the key in question.

`On_Terminal : Terminal;`
Specifies the particular terminal for which the key should be interpreted.

The following terminal names are currently supported:

- Rational
- VT100

`return String;`
Returns the name of the specified key for the specified terminal.

Example

This function is especially effective when used in conjunction with the System_Utilities.Terminal_Type function (SMU). This function returns a string image for the currently connected terminal.

```
    A_Key : Raw.Key;
    Character_Stream  : Raw.Stream_Type;

begin
...
    Raw.Get (Stream => Character_Stream,  Key => A_Key);
    if Raw.Image (A_key,
                  System_Utilities.Terminal_Type)  = "F1" then
    ...
    elsif Raw.Image (A_key,
                  System_Utilities.Terminal_Type)  = "F2" then
    ...
    end if;
```

type Key

```
type Key is new Natural range 0 .. 1023;
```

Description

Defines the possible range of keys.

type Key_String

```
type Key_String is array (Positive range <>) of Key;
```

Description

Defines an unconstrained array type for use in holding a series of keys.

procedure Open

```
procedure Open (Stream : in out Stream_Type);
```

Description

Disconnects keystroke input from the Rational Editor and opens the keystroke stream for use by the currently executing job.

Parameters

```
Stream :  in out Stream_Type;
```

Specifies the handle for the stream.

Errors

If the stream is already open, the Status_Error exception is raised.

If the caller is not the current job—that is, the user or program interrupted with a Job.Disconnect (SJM)—the Io_Exceptions.Status_Error exception is raised.

Example

One method of using this procedure is:

```
    A_Key : Raw.Key;
    Quit_Key : constant Raw.Key := ...;

    The_Stream : Raw.Stream_Type;
...
begin
    Raw.Open (The_Stream);
    loop
        Raw.Get (The_Stream, A_Key);
        if A_Key = ... then
            ...
        elsif A_Key = ... then
            ...
        elsif A_Key = Quit_Key then
            Raw.Close (The_Stream);
            exit;
        else
            ...
    end loop;
```

subtype Simple_Key

```
subtype Simple_Key is Key range Ø .. 127;
```

Description

Defines the allowable range for simple keys.

Simple keys correspond to the 128 ASCII characters as defined in PT, package Standard. The value of the simple keys corresponds to the 'Pos attribute of the Character type.

type Stream_Type

```
type Stream_Type is private;
```

Description

Defines a handle for access to the keystroke stream.

subtype Terminal

```
subtype Terminal is String;
```

Description

Defines a subtype string for holding terminal names.

The following terminal names are currently supported:

- Rational
- VT100

exception Unknown_Key

```
Unknown_Key : exception;
```

Description

Defines an exception raised by the Value function if the specified key name does not have a corresponding key for the specified terminal.

References

function Value

function Value

```
function Value (For_Key_Name  : String;
                On_Terminal   : Terminal) return Key;
```

Description

Returns the corresponding key for a key name defined in package !Machine.Editor_Data.Visible_Keynames.

Parameters

`For_Key_Name : String;`

Specifies the string image of the key.

`On_Terminal : Terminal;`

Specifies the terminal for which the key mapping is desired.

`return Key;`

Returns the key corresponding to the key name.

Errors

If the specified key name does not have a corresponding key for the specified terminal, the Unknown_Key exception (in this package) is raised.

Example

The following code could be used to define named constants for the arrow keys:

```
with System_Utilities,Raw;

package Key_Definitions is

    Up    : constant Raw.Key := Raw.Value ("UP",
                                System_Utilities.Terminal_Type);
    Down  : constant Raw.Key := Raw.Value ("DOWN",
                                System_Utilities.Terminal_Type);
    Left  : constant Raw.Key := Raw.Value ("LEFT",
                                System_Utilities.Terminal_Type);
    Right : constant Raw.Key := Raw.Value ("RIGHT",
                                System_Utilities.Terminal_Type);

end Define_Keys;
```

procedure Value

```
procedure Value (For_Key_Name  :      String;
                 On_Terminal   :      Terminal;
                 Result        : out Key;
                 Found         : out Boolean);
```

Description

Provides the corresponding key for a key name defined in package !Machine.Editor_Data.Visible_Keynames.

Parameters

`For_Key_Name : String;`

Specifies the string image of the key.

`On_Terminal : Terminal;`

Specifies the terminal for which the key mapping is desired.

`Result : out Key;`

Specifies the key corresponding to the key name.

`Found : out Boolean;`

Specifies whether the value in the Result parameter is valid (that is, whether the specified key name has a corresponding key for the specified terminal).

Example

The following code could be used to define named constants for the arrow keys:

```
package Define_Keys is
    Up    : Raw.Key;
    Down  : Raw.Key;
    Left  : Raw.Key;
    Right : Raw.Key;
end Define_Keys;

package body Define_Keys is;

    Found : Boolean;

    Not_Successful : exception;

    procedure Assert_Success (Found : Boolean) is
    begin
        if not Found then
            raise Not_Successful;
        end if;
    end Assert_Success;

begin

    Raw.Value ("UP",
               System_Utilities.Terminal_Type, Up, Found);
    Assert_Success (Found);

    Raw.Value ("DOWN",
               System_Utilities.Terminal_Type, Down, Found);
    Assert_Success (Found);

    Raw.Value ("LEFT",
               System_Utilities.Terminal_Type, Left, Found);
    Assert_Success (Found);

    Raw.Value ("RIGHT",
                System_Utilities.Terminal_Type, Right, Found);
    Assert_Success (Found);

exception
    when Not_Successful =>
        ...
end Define_Keys;
```

end Raw;

```
end Window_Io;
```

Index

This index contains entries for each unit and its declarations as well as definitions, topical cross-references, exceptions raised, errors, enumerations, pragmas, switches, and the like. The entries for each unit are arranged alphabetically by simple name. An italic page number indicates the primary reference for an entry.

B

C

D

E

F

J

K

L

M

N

P

R

S

T

U

V

W

RATIONAL

READER'S COMMENTS

Note: This form is for documentation comments only. You can also submit problem reports and comments electronically by using the SIMS problem-reporting system. If you use SIMS to submit documentation comments, please indicate the manual name, book name, and page number.

Did you find this book understandable, usable, and well organized? Please comment and list any suggestions for improvement.

_____
_____
_____
_____
_____
_____

If you found errors in this book, please specify the error and the page number. If you prefer, attach a photocopy with the error marked.

_____
_____
_____
_____

Indicate any additions or changes you would like to see in the index.

_____
_____
_____
_____

How much experience have you had with the Rational Environment?

6 months or less _____ 1 year _____ 3 years or more _____

How much experience have you had with the Ada programming language?

6 months or less _____ 1 year _____ 3 years or more _____

Name (optional)_____ Date_____
Company _____
Address _____
City _____ State _____ ZIP Code _____

Please return this form to:

Publications Department
Rational
1501 Salado Drive
Mountain View, CA 94043